MASTER EN SCIENCES INFORMATIQUES

Le problème des horaires : analyse de moyens de résolution récents

Simonis, Emmanuel

Award date:
1989

Awarding institution:
Universite de Namur

Année académique
1988-1989

Facultés Universitaires Notre-Dame de la Paix
Namur
Institut d'informatique

Le problème des horaires : analyse de moyens de résolution récents

Emmanuel SIMONIS

Mémoire présenté pour l'obtention du grade de
Licencié en Sciences, option Informatique

Promoteur : Jean-Paul LECLERCQ

Remerciements

Je remercie vivement Jean-Paul Leclercq
pour son assistance et ses précieux conseils,
ainsi que Jean-Marie Jacquet et Bernard Detrembleur
pour leur aide quant à l'utilisation
du C-Prolog et d'Alma.
Je tiens aussi à remercier Katrien
pour son soutien et sa compréhension.

TABLE DES MATIERES

Table des matières ..... page 3

Chapitre 1

INTRODUCTION

1.1 Objectifs

Au départ, ce mémoire avait pour seul objectif l'analyse de l'utilisation de la programmation logique, et en particulier du langage Prolog, pour construire des horaires de cours. Cette analyse devait porter plus sur la simple faisabilité que sur l'efficacité. Ainsi, quelques contraintes seulement devaient être étudiées, pour mettre à l'épreuve la capacité d'adaptation du langage. La convivialité espérée d'un langage comme Prolog, au niveau de la saisie des données du moins, était plus intéressante à nos yeux que la plus ou moins grande rapidité de résolution du problème. En effet, la construction "manuelle" d'un horaire prend beaucoup de temps, souvent plusieurs jours, de sorte qu'un programme effectuant ce travail (même en quelques heures), sans demander à l'utilisateur de manipulation complexe, nous paraît intéressant.

En cours de réalisation, craignant une impasse suite à de mauvais résultats obtenus (peut-être dus en partie à une mauvaise approche de la programmation logique, entièrement nouvelle pour nous), nous avons prévu la possibilité d'élargir le sujet d'étude. Comme on le verra, un modèle classique pour les problèmes d'horaire est la coloration de graphe. Il nous a paru intéressant de considérer des méthodes de coloration assez récentes, qui semblent offrir plus de possibilités grâce à l'usage d'une fonction objectif (à optimiser).

Cependant, grâce à une meilleure compréhension de la programmation logique, et à une approche différente du problème, les résultats obtenus se sont améliorés et nous avons préféré poursuivre dans cette voie, laissant l'approche *coloration par fonction objectif* au second plan. Ce second modèle occupe de ce fait une part assez réduite de ce mémoire ; nous présentons seulement quelques réflexions sur la construction de la fonction objectif, mais pas d'implémentation.

1.2 Qu'attendions-nous de Prolog ?

A priori, avant même d'apprendre le Prolog, nous avions les espoirs suivants quant aux possibilités offertes par ce langage.

1. La programmation logique demandant, pensions-nous, une simple description du problème, les horaires devaient être particulièrement faciles à construire, en particulier via leur modélisation sous forme de graphe à colorer.

En effet, le problème de coloration est fort simple à décrire. Il consiste à attribuer une couleur à chaque sommet, de telle façon que deux sommets reliés par une arête aient une couleur différente, et ce en utilisant un minimum de couleurs. Nous espérions donc pouvoir colorer un graphe avec une seule règle, ressemblant à celle-ci :

couleur(Sommet,Couleur) $\Leftarrow$
$\neg$(adjacents(Sommet,Somm_Adj) $\wedge$ couleur(Somm_Adj,Couleur)).

2. En outre, nous attendions de Prolog qu'il permette la saisie des données dans un langage pseudo-naturel, sous forme de clauses (comme "Dupont est un professeur." ou "Dupont ne peut donner cours le mardi de 14 à 16 heures."), sans impliquer la programmation d'une interface complexe, comme ce serait le cas dans un langage traditionnel.

3. Enfin, il nous semblait que les diverses contraintes que l'on peut rencontrer en pratique dans un problème d'horaire seraient plus facilement manipulables en Prolog, à l'aide de règles exprimant ces contraintes.

Qu'en est-il en réalité ?

1. Une fois le langage appris, il s'est avéré que le premier de ces espoirs était largement surfait. En effet, en Prolog comme partout, une définition du problème en termes de sa solution n'est pas un outil de résolution. Pour appliquer la règle selon laquelle un sommet ne peut être de même couleur que ses adjacents, il faut connaître la couleur de ceux-ci, c'est-à-dire connaître la solution. Si cette règle suffit à décrire le problème, elle ne suffit pas à le résoudre.

2. Par contre, la saisie des données en langage à peu près naturel est, elle, réalisable. Mais ce sera au prix d'une syntaxe précise à respecter et de l'utilisation fréquente du caractère de soulignement, nécessaire pour former des atomes Prolog ("mots" du langage). Comme nous le verrons, on peut facilement amener un programme Prolog à accepter, sans autre manipulation, des données du genre "dupont est_un_professeur." ou "dupont est_indisponible_pour mardi_14h.".

3. Enfin, les contraintes pourront être directement représentées par des procédures les validant. Simplement, l'usage de ces procédures dépendra de la

façon dont les contraintes seront prises en charge : vérification a posteriori ou instrument de choix.

1.3 Plan

Outre cette introduction, ce mémoire est structuré comme suit.

Le chapitre 2 définit le problème des horaires de base et les quelques extensions que nous en étudierons.

Le chapitre 3 décrit les moyens de résolutions qui nous intéressent : le modèle de la coloration de graphes et l'outil de la programmation logique.

Dans le chapitre 4, nous ferons quelques propositions quant à la modélisation d'un problème d'horaire sous forme d'un graphe à colorer.

Le chapitre 5 contient les descriptions de notre travail d'implémentation, en langage Prolog.

Nos conclusions clôturent ce premier volume.

Le second volume, en annexe, contient les listings commentés et des résultats de tests.

Chapitre 2

LE PROBLEME DES HORAIRES

2.1 Problème de base

Dans ce mémoire, nous considérerons principalement le problème des horaires *de cours*, surtout dans le cadre des écoles secondaires, où il se pose de façon assez marquée et où il est sans doute plus difficile à résoudre (plus de cours et de classes pour moins de professeurs) qu'à l'université. A titre de variante, nous abordons plus loin le problème des horaires *d'examen*, pour en relever quelques particularités.

On peut définir comme suit un problème d'horaire de cours (modèle de base[1]).

Etant donnés :

- la grille horaire, c'est-à-dire la liste des heures de la semaine[2],
- la liste des heures de cours, avec pour chacune un nom, le nom d'un ou de plusieurs professeurs et le nom d'une ou plusieurs classes d'élèves, et
- la liste des indisponibilités de chaque professeur, c'est-à-dire la liste des heures de la grille pendant lesquelles ce professeur ne peut donner cours,

il faut attribuer à chaque heure de cours une position dans la grille de sorte que :

- un professeur ne donne qu'un cours à la fois ;
- une classe ne suive qu'un cours à la fois ;
- un professeur ne donne jamais cours lorsqu'il est indisponible.

On peut y ajouter un certain nombre d'extensions.

---

[1] Il est important de noter que nous incluons dans le modèle de base les contraintes de disponibilité des professeurs. Celles-ci sont en effet très importantes et, suite aux propositions de Henkes [12], assez facilement modélisables.

[2] ou, plus généralement, des plages horaires.

2.2 Extensions

2.2.1 Introduction

Nous distinguons deux façons de généraliser le problème de base. La première consiste à imposer de nouvelles *contraintes* aux grilles à construire. La seconde est l'extension de l'*objectif* à atteindre, du travail de recherche à effectuer.

On peut rencontrer des contraintes de natures fort diverses. Nous n'en étudierons ici que de trois types : plage imposée pour un cours, groupe de cours simultanés et paires de cours consécutifs.

Un placement imposé concerne une seule heure de cours, et ne lui permet qu'une position dans la grille.

Une contrainte de simultanéité met en relation plusieurs cours, indépendamment d'une plage horaire particulière, et a donc autant de solutions qu'il n'y a de positions dans la grille.

La consécutivité contraint les cours deux par deux, en relation avec la grille : elle nécessite la définition des plages horaires consécutives. Elle permet en outre des combinaisons plus complexes, reliant plusieurs cours dont les places dans l'horaire seront interdépendantes.

Cet échantillon de contraintes, toutes assez utiles et souvent rencontrées, nous semble suffisant pour donner une idée des techniques que l'on pourra ou non appliquer à un problème d'horaire très général. Elles présentent en outre une certaine complémentarité avec celles du problème de base, du point de vue de leur prise en charge dans un programme non-déterministe, comme on le verra au chapitre 5.

Nous examinerons d'autre part deux extensions "fonctionnelles" du modèle : la recherche d'un horaire présentant un minimum d'heures de fourche pour les professeurs, et l'attribution des locaux en plus des plages horaires.

Le problème des heures de fourche permet une bonne illustration des difficultés que l'on peut rencontrer dans les extensions de type optimisation.

Quant à l'attribution des locaux, son importance pratique suffit à ce que nous nous y intéressions.

2.2.2 Contraintes supplémentaires

2.2.2.1 Attribution d'une plage a priori

La possibilité d'imposer une plage à un cours nous semble fort importante. D'abord, cette contrainte correspond à des situations réelles : activités liées à des horaires indépendants (telles que conférences ou télévision scolaire), intervention d'une personne extérieure. Mais en plus, elle permet d'effectuer la perturbation d'une solution déjà trouvée ou d'un horaire existant, que l'on voudrait modifier tout en conservant tels quels certains choix.

Nous appellerons cette contrainte "attribution d'une plage a priori" ou, plus brièvement, "placement imposé".

2.2.2.2 Contrainte de simultanéité de cours

Dans les horaires des écoles secondaires, on trouve souvent des exigences du genre "les cours de gymnastique des garçons et des filles de la même année doivent se donner en même temps", d'application aussi pour les cours de religion et de langues, où la séparation de la classe se fait selon la confession ou les langues choisies.

Pour ce genre de contrainte, nous étudierons la possibilité d'introduire une contrainte de simultanéité d'un ensemble donné de cours, auxquels devra être associée la même position dans la grille horaire.

On voit ici que la modélisation d'un problème d'horaire implique une notion de classe réduite à un groupe d'élèves qui suivent exactement les mêmes cours. Ainsi, selon la découpe sexe-confession-langues, une classe pourra être éclatée en huit[3] "objets" à manipuler.

2.2.2.3 Problème des cours consécutifs

On trouve souvent en pratique des cours de durée supérieure à la normale. Par exemple, le professeur de gymnastique pourra demander deux heures successives pour aller au bassin de natation. Autre situation, plutôt universitaire : une séance de travaux pratiques que l'on voudrait voir suivre le cours théorique correspondant.

---

[3] ou plus, selon la diversité de choix pour les cours confessionnels et les langues.

On étudiera donc la possibilité d'imposer à deux cours d'être donnés à la suite l'un de l'autre. Ceci nécessite la définition de périodes consécutives de la grille horaire : les donner dans l'ordre chronologique ne suffit pas. En effet, la première heure d'une journée ne suit pas immédiatement la dernière de la veille, et la première heure de l'après-midi ne suit pas la dernière de la matinée. Sans parler des récréations !

2.2.3 Fonctions supplémentaires

2.2.3.1 Problème des heures de fourche

Le problème des heures de fourches est assez complexe[4]. Sa définition elle-même n'est pas universelle.

Définition :

Une heure de la grille horaire est dite **heure de fourche** d'un professeur *SSI* ce professeur ne donne pas cours pendant cette heure, mais donne, dans la même journée, au moins un cours avant et un cours après cette heure.

Cette définition nous semble bien correspondre à l'intuition[5], bien que chaque professeur en ait sans doute une conception plus personnelle.

Quels objectifs peut-on rechercher ? Nous pouvons en imaginer plusieurs. Citons principalement :

- Minimisation du nombre total d'heures de fourche.
- Minimisation du nombre d'heures de fourche du professeur qui en a le plus (minimax).
- Minimisation de la somme des dépassements des nombres d'heures de fourche acceptés par chaque professeur.

[4]Les heures de fourche font d'ailleurs une des grandes faiblesses des algorithmes de création automatique d'horaires : ils satisfont les contraintes posées, mais ne donnent pas de "beaux" horaires.

[5]Rappelons que nous nous situons dans le cadre des écoles secondaires, où les professeurs ont plus de cours qu'à l'université.

Du point de vue de la résolution, tous sont similaires : ils s'expriment sous forme d'un problème d'optimisation d'une fonction entière bien définie. Nous nous limiterons donc au premier objectif.

2.2.3.2 Problème des locaux

En pratique, un horaire au complet attribue à chaque heure de cours non seulement une position dans la grille horaire, mais aussi un local, de sorte qu'à deux cours se donnant en même temps ne soit pas attribué le même local. On peut aussi prendre en compte un grand nombre de contraintes supplémentaires. Nous n'en retiendrons que deux, susceptibles de faire partie d'un *problème de base* pour les locaux : la capacité du local et la "localisation" a priori (imposant un local à un cours). Mais on pourrait, par exemple, proposer les contraintes suivantes :

- attribuer des locaux à des professeurs ou à des classes ;
- demander que deux cours aient lieu dans le même local ;
- imposer ou interdire un local à un cours ou, de façon plus générale, une liste de locaux à une liste de cours (pour permettre la représentation de la spécificité de certains locaux : salles de gymnastique, laboratoires de langues ou de sciences, etc) ;
- interdire l'occupation d'un local à certains moments (pour en permettre l'entretien, par exemple) ;
- demander qu'un cours ne reçoive pas de local (ce qui est nécessaire à la prise en compte d'activités extérieures à l'école).

2.3 Variante : les horaires d'examen

Le problème des horaires peut aussi s'appliquer aux horaires d'examen, en particulier à la planification de sessions de type universitaire. Le principe est le même que pour les cours, mais avec certaines particularités.

- La grille horaire est plus étendue : plusieurs semaines (sans répétition).

- La grille n'est pas découpée en heures, comme c'est généralement le cas pour les cours. La découpe dépend du point de vue utilisé : horaire du professeur (découpe en demi-journées) ou horaire de l'étudiant (découpe en fractions d'heures, par exemple de 15 minutes, pour planifier les examens oraux).

- Le groupement d'étudiants ne se base plus sur la classe. Pour les oraux, il pourra se faire par très petits ensembles de un ou de quelques étudiants.

En outre, certaines extensions ne sont plus d'application, ou s'utilisent autrement. Ainsi, la simultanéité d'examens ne semble pas avoir grand intérêt. Par contre, les examens pouvant avoir des durées très diverses, une contrainte de consécutivité d'examens serait très utile. On notera aussi que le problème des "heures" de fourche est complètement modifié. D'abord, on ne comptera plus par heures, mais plutôt par demi-journées : il faudra une définition très différente de la "demi-journée de fourche". Ensuite le problème est complètement inversé pour les étudiants : ceux-ci apprécieront grandement d'avoir un peu d'espacement entre leurs examens.

Enfin, le problème dans son ensemble est sans doute moins critique pour les professeurs que pour les étudiants, à l'inverse de l'horaire de cours.

Toutes ces différences plaident pour la réalisation de deux applications distinctes autour d'un même noyau, plutôt que d'un seul programme traitant les deux types d'horaires.

2.4 Conventions de vocabulaire

Afin d'éviter les ambiguïtés auxquelles ont peut s'attendre avec les mots "classe" et "heure", nous prenons les conventions suivantes.

(**Heure de**) **cours** : période indivisible durant laquelle un professeur donne une séance d'un cours donné à une classe donnée. (Exemple : s'il existe deux classes suivant séparément un même cours de mathématiques, donné par le même professeur, à raison de 7 heures par semaine, cela représente 14 heures de cours.)

Remarque : pour abréger, nous parlerons en général de "cours" au lieu de "heure de cours".

Plage (horaire) : période indivisible (de même durée que les heures de cours) de la grille horaire. (Exemple : l'horaire d'une école, ouverte 5 jours par semaine et où sont prévues 8 heures de cours chaque jour sauf le mercredi (4 heures seulement), compte 36 plages horaires.)

Classe : groupe d'élèves ayant toujours cours ensemble (ou considérés comme tels dans la modélisation).

Local : lieu où peut se donner un cours ou une activité considérés dans l'horaire.

Chapitre 3

MOYENS DE RESOLUTION

3.1 Un modèle : la coloration de graphes

3.1.1 Principe général

Remarque : Pour les notions de base de la théorie des graphes, nous renvoyons le lecteur aux ouvrages sur le sujet, et notamment [5,6].

Soit $G = (X, V)$ un graphe non-orienté (l'ensemble des sommets de G est X, l'ensemble de ses arêtes, ou paires de sommets, est V). Soit C un ensemble de couleurs.

Définitions :

Une **coloration** de G est une fonction

$$\begin{aligned} h \ : \ & X \to C \\ & x \rightsquigarrow h(x) \end{aligned}$$

qui à tout sommet x associe une couleur $h(x)$ de telle façon que

$$\forall x_1, x_2 \in G, \ \overline{(x_1, x_2)} \in V \ \Rightarrow \ h(x_1) \neq h(x_2).$$

Autrement dit, colorer un graphe consiste à donner une "couleur" à chacun de ses sommets, de telle façon que deux sommets adjacents aient une couleur différente.

Nous appelons **coloration impropre** de G toute fonction $h : X \to C$.

On cherche généralement à colorer un graphe en un nombre minimum de couleurs (C de taille minimale) ou en un nombre donné de couleurs (C de taille fixée).

Le premier problème est résolu par des méthodes que nous qualifierons de "classiques". Notons qu'il est très difficile de trouver une coloration minimale d'un graphe : il s'agit d'un problème NP-complet. On se contente généralement d'algorithmes heuristiques, plus simples mais donnant seulement une solution proche de l'optimum.

Nous traiterons le second problème par des méthodes basées sur la minimisation d'une fonction objectif, et n'utilisant pas le principe habituel d'amélioration itérative.

3.1.2 Deux méthodes "classiques"

3.1.2.1 Introduction

Parmi les nombreuses méthodes heuristiques de coloration, nous en avons choisi deux, LF et CSG, l'une pour sa simplicité, l'autre pour son adéquation aux graphes structurés correspondant à la plupart des problèmes pratiques.

Ces deux méthodes, décrites en détail dans les sections suivantes, ont pour objectif de trouver une bonne coloration d'un graphe donné, mais ne garantissent pas de trouver la coloration optimale.

En outre, elles sont déterministes, en ce sens qu'elles ne donnent qu'une seule coloration d'un graphe. En effet, leur algorithme ne laisse pas (ou très peu) de liberté aux décisions et, lorsqu'un choix est possible, n'en explorent qu'une seule branche. Notons cependant que cette branche pourrait être sélectionnée aléatoirement, mais ce serait au prix d'une certaine perte d'efficacité (par exemple, la recherche d'un sommet de degré maximal devrait donner tous les "candidats", et non plus seulement l'un d'entre eux). L'usage du hasard ne permettrait pas aux méthodes classiques de donner plusieurs solutions par exécution, mais au moins pourraient-elles trouver différentes coloration d'un graphe.

3.1.2.2 La méthode "Largest First"

3.1.2.2.1 Principe et algorithme

La méthode "Largest First" (ou "LF") consiste à colorer les sommets par ordre de degré décroissant. Ainsi, on colore d'abord les sommets les plus difficiles à colorer, c'est-à-dire ceux qui ont le plus d'adjacents.

Algorithme :

Tant qu'il reste un sommet non-coloré :
- choisir un sommet non-coloré x de degré maximal,
- donner à x la première[1] couleur qui ne lui est pas interdite.

[1]En pratique, les couleurs sont désignées par des entiers, à partir de 1. L'expression "première couleur" fait référence à cette représentation, où les couleurs ayant les plus petits numéros sont les premières. Trouver la première couleur non interdite, cela revient donc à boucler sur les couleurs (donc sur les entiers), à partir de la première, jusqu'à en trouver une qui ne soit pas présente parmi les adjacents du sommet.

3.1.2.2.2 Propriétés

Cette méthode présente l'inconvénient de produire une mauvaise répartition des couleurs, les "premières" étant généralement plus représentées. En outre, les colorations produites sont relativement éloignées de l'optimum, utilisant en moyenne 28 % de couleurs en trop pour de grands graphes de Leighton.

Elle est par contre fort intéressante au point de vue de la complexité : elle est polynomiale de complexité théorique égale à 2 et de complexité pratique égale à 1,7.

(Ces résultats proviennent de [10].)

3.1.2.3 La méthode "Complete Sub-Graph"

3.1.2.3.1 Définitions et propriétés

La méthode "Complete Sub-Graph" (ou "CSG") est nettement plus complexe, mais donne de très bons résultats de coloration pour les graphes structurés. Elle se base sur une propriété des sous-graphes complets que nous rappelons ici.

Définitions :

Soit $G = (X,V)$ un graphe non-orienté.

Un **sous-graphe** de G est un graphe $H = (Y,W)$ tel que

$$Y \subseteq X$$
$$et \quad W = \left\{ \overline{(y_1, y_2)} \in V \ \dagger \ y_1, y_2 \in Y \right\}$$

C'est donc un graphe H ayant pour sommets une partie des sommets de G et pour arêtes celles de G qui relient deux sommets de H.

Un graphe **complet** est un graphe $G = (X,V)$ tel que

$$\forall x_1, x_2 \in X, \ x_1 \neq x_2, \text{ on a } \overline{(x_1, x_2)} \in V.$$

Autrement dit, c'est un graphe dont chaque sommet est adjacent à tous les autres.

Une **clique** de G est un sous-graphe complet de G.

Propriété :

Une clique de n sommets doit être colorée en n couleurs.

En effet, chaque sommet étant adjacent à tous les autres, deux sommets ne pourront jamais avoir la même couleur. Il faudra donc une couleur différente pour chacun des n sommets, soit n couleurs.

Conséquence :

Si un graphe contient une clique de k sommets, alors il ne pourra être coloré en moins de k couleurs.

3.1.2.3.2 Principe et algorithme

La méthode CSG consiste à trouver une clique aussi grande que possible dans le graphe à colorer, d'associer une couleur différente à chacun des sommets de cette clique et de colorer le reste du graphe en commençant par le sommet qui a le plus de couleurs interdites.

Algorithme :

Soit G un graphe à colorer. L'algorithme CSG est le suivant.

- Rechercher une grande clique C de G (par exemple, avec l'heuristique suivante).
  - Choisir un sommet de degré d'ordre 2 maximal. Soit x_1 ce sommet, soit Y_1 l'ensemble de ses adjacents. Prendre $C = \{x_1\}$.
  - Former le sous-graphe H_1 engendré par l'ensemble de sommets Y_1. Prendre $i = 1$.
  - Tant que H_i n'est pas vide,
    - Choisir un sommet de degré maximal dans H_i. Soit x_{i+1} ce sommet, soit Y_{i+1} l'ensemble de ses adjacents. Prendre $C = C \cup \{x_{i+1}\}$.
    - Former le sous-graphe H_{i+1} engendré par l'ensemble de sommets Y_{i+1}.
    - Incrémenter i de 1.
- Colorer les sommets de la clique C, en $k = i$ couleurs.
- Colorer le reste du graphe :

  Tant qu'il reste un sommet non coloré :

- Choisir un sommet non-coloré x dont le nombre de couleurs interdites est maximal. Soit l ce nombre.
- Selon les valeurs de l et k : [2]

  Si $l = k - 1$, donner à x la seule couleur possible.

  Si $l < k-1$, donner à x la couleur la plus interdite aux adjacents non-colorés de x (de sorte à minimiser le nombre d'interdictions nouvelles sur les adjacents non-colorés de x).

  Si $l = k$, donner à x la couleur $k+1$. Incrémenter k de 1.

3.1.2.3.3 Propriétés de la méthode

La méthode CSG effectue une répartition assez équilibrée des couleurs dans le graphe. Elle est en outre bien adaptée aux graphes structurés, en particuliers les graphes modélisant les horaires, car leur nombre chromatique est proche de la taille de la plus grande clique (ce qui n'est pas le cas des graphes aléatoires). Elle colore les graphe de Leighton avec 12 % de couleurs en trop, en moyenne.

Elle présente par contre une complexité nettement plus élevée que LF : polynomiale de complexité théorique égale à 3 et de complexité pratique égale à 2.8 (sur les graphes de Leighton).

(Ces résultats sont donnés dans [10].)

3.1.3 Deux méthodes avec fonction objectif

3.1.3.1 Introduction

Les méthodes que nous décrivons ci-dessous sont d'application générale en recherche opérationnelle. Nous verrons qu'elles peuvent s'appliquer en particulier à notre problème, en répondant à l'objectif de colorer un graphe en un nombre donné de couleurs. En outre, elles sont non-déterministes en ce sens que, même si elles ne peuvent fournir qu'une solution par exécution, deux exécutions successives pourront donner des résultats totalement différents.

Leur principe diffère fondamentalement de celui des méthodes de coloration "classiques". Elles reposent en effet sur des techniques très récentes[3]

[2] On n'aura jamais $l > k$, car x ne peut pas avoir plus de couleurs interdites qu'il n'y en a dans le graphe.

[3] Le recuit simulé ("simulated annealing") est apparu en 1982, le "tabu-search" en 1985.

d'optimisation. Ces techniques, abandonnant les sentiers battus de l'amélioration itérative acceptent une dégradation contrôlée de la fonction objectif, permettant de sortir de la cuvette d'un optimum local. De plus, elles travaillent sur des fonctions qui doivent seulement être évaluables, sans aucune exigence de dérivabilité d'aucun ordre.

Pour appliquer cette idée à la coloration de graphes, il fallait définir une fonction à optimiser. Voici celle que proposent Hertz et de Werra [13].

Soit un graphe $G = (X, V)$ à colorer en k couleurs.

La fonction objectif f est définie sur l'ensemble des colorations impropres de G, c'est-à-dire des partitions des sommets de G en k ensembles $C_1, \ldots, C_k$, chacun correspondant à une couleur. Elle compte le nombre de violations à la règle de coloration, c'est-à-dire le nombre d'arêtes reliant deux sommets du même ensemble (donc de même couleur).

Autrement dit :

$$\begin{aligned} f \ : \ & \left\{ (C_1, \ldots, C_k) \ \dagger \ X = C_1 \dot{\cup} \ldots \dot{\cup} C_k \right\} \longrightarrow \mathbb{N} \\ & (C_1, \ldots, C_k) \rightsquigarrow \textstyle\sum_{i=1}^{k} \# \left\{ \overline{(x_1, x_2)} \in V \ \dagger \ x_1, x_2 \in C_i \right\} \end{aligned}$$

Soit une coloration impropre initiale (à définir). Elle va être modifiée progressivement dans l'espoir de voir décroître f. En effet, lorsque cette fonction vaut 0, il n'existe plus d'arêtes dont les deux sommets appartiennent au même ensemble C_i. Par conséquent, si l'on donne la couleur i à tous les sommets de l'ensemble C_i, $i = 1 \ldots k$, on obtient une coloration de G en k couleurs.

Nous décrivons rapidement ci-dessous les méthodes d'optimisation en question.

3.1.3.2 La méthode "Simulated Annealing" (1982)

La méthode du "recuit simulé" (ou "simulated annealing") utilise pour minimiser une fonction objectif f le principe suivant. (On en trouvera une description plus complète dans [7] ou [8]).

Au départ d'un état E, où l'objectif vaut $f(E)$, on choisit aléatoirement une transformation vers un état voisin E' (la notion de voisinage étant à définir).

On décide alors si l'on poursuit à partir de E' ou à partir de E :

$$\text{on conserve } E' \text{ avec une probabilité } \begin{cases} 1 & \text{si } f(E') \leq f(E), \\ \exp\left(-\frac{f(E')-f(E)}{T}\right) & \text{si } f(E') > f(E). \end{cases}$$

La "température" T est un paramètre permettant de contrôler l'amplitude des mouvements "montants" (qui évitent le blocage dans les minima locaux). Il est élevé au départ et décroît géométriquement au fur et à mesure des itérations.

L'algorithme est caractérisé par différents paramètres. D'une part, la température initiale, le nombre d'itérations entre chaque baisse de température, le facteur de décroissance de celle-ci, le critère d'arrêt (la température minimale). D'autre part, la règle de production de l'état initial, la règle de transformation des états (définition d'états voisins).

Nous verrons dans le chapitre suivant comment définir ces derniers éléments dans le contexte de la coloration de graphes. (Ces définitions s'appliqueront également à la seconde méthode.)

3.1.3.3 La méthode "Tabu-Search" (1985)

Le principe de la méthode "Tabu-Search" est plus simple. (Nous le décrivons ici brièvement ; nous renvoyons à [9] pour plus d'informations.)

Au départ d'un état E de "valeur" $f(E)$, on génère v états voisins $E_1, \ldots, E_v$. Soit E_i celui de ces états qui a la valeur la plus basse :

$$f(E_i) = \min_{j=1,\ldots,v} f(E_j).$$

Que $f(E_i)$ soit ou non inférieur à $f(E)$, on reprendra à partir de E_i ($E := E_i$), et ainsi de suite jusqu'à avoir effectué un nombre d'itérations fixé[4]. Mais auparavant, on ajoute à une *liste de tabous* la transformation qui fait passer de E_i à E. Cette liste, de longueur fixée, est consultée chaque fois que l'on génère une transformation, celle-ci étant rejetée si elle apparaît dans la liste. On s'interdit ainsi de revenir sur ses pas, ce qui doit permettre de sortir d'un minimum local par un "col" d'une certaine hauteur.

[4] Si l'on connaît le minimum, comme c'est le cas pour la coloration de graphes, on peut s'arrêter dès qu'il est atteint.

En outre, on retient en permanence le meilleur état visité.

Cet algorithme est caractérisé par les paramètres suivants : le nombre d'itérations, le nombre de voisins générés à chaque itération, et la taille de la liste de tabous. Les valeurs optimales de ces paramètres peuvent varier selon le type de problème. Notons cependant que Glover [9], qui a proposé la méthode, suggère d'utiliser une liste de 7 tabous : au-delà, on n'améliore plus la procédure, tout en augmentant le temps de calcul.

En outre, il faut fixer la procédure de génération d'un état initial, et définir les transformations de voisinage. Ceci aussi dépend du problème traité ; nous allons voir dans le chapitre 4 comment on peut procéder en coloration de graphes.

3.2 Un outil : la programmation logique

Dans son "Introduction to Logic Programming" [2], Hogger écrit notamment ce qui suit.

> La programmation logique est l'usage de la logique comme langage de programmation. Cet usage relativement récent de la logique est encore méconnu de la plupart des informaticiens. Mais l'intérêt pour la programmation logique est stimulé par le rôle de principal formalisme pour la prochaine génération d'ordinateurs qu'on lui attribue généralement.
>
> La programmation logique est fondamentalement différente de la programmation "conventionnelle" en ce sens qu'elle demande une description de la structure logique des problèmes plutôt qu'une description de la façon de les résoudre.
>
> Les personnes qui font connaissance de l'informatique via un langage de programmation classique sont souvent surpris de s'apercevoir que la programmation n'est pas une simple question de logique, issue directement de la conception première du problème, mais paye un lourd tribut aux mécanismes internes de l'ordinateur.
>
> Inversément, les programmeurs habitués aux seuls langages traditionnels peuvent avoir le même genre de problèmes en s'initiant à

la programmation logique. Instinctivement inquiets de contrôler la machine efficacement, ils éprouvent un vague sentiment de privation lorsqu'ils travaillent avec un langage qui ne permet pas ce contrôle.

Ainsi, un effort important est nécessaire pour se défaire d'un vieux conditionnement à une perception de la programmation.

On compte principalement deux langages de programmation logique : LISP et PROLOG. C'est en PROLOG que nous avons travaillé, et plus précisément avec la version 1.5 de C-Prolog, sur VAX/UNIX.

Chapitre 4

IDEES DE MODELISATION

4.1 Problème de base

4.1.1 Introduction

Le problème d'*horaires de cours* de base peut assez facilement se modéliser sous forme d'un problème de coloration de graphe. Nous utilisons pour cela le graphe défini comme suit, selon la proposition de Henkes [12].

Sommets :
- Associer un sommet à chaque heure de cours.
- Associer un sommet à chaque plage de la grille horaire.

Arêtes :
- Relier par une arête toute paire de sommets-*plages horaires*.

(Conséquence : chaque plage horaire aura une couleur différente.)

- Relier par une arête toutes les paires de sommets associés à des cours donnés par le même professeur ou suivis par la même classe.

(Conséquence : deux cours donnés par le même professeur ou suivis par la même classe auront une couleur différente.)

- Relier par une arête toute paire (sommet-*plage horaire*, sommet-*cours*) telle que le professeur donnant ce cours est indisponible pendant cette plage horaire.

(Conséquence : un cours donné par un professeur indisponible à une plage horaire donnée n'aura pas la couleur associée à cette plage.)

Si l'on parvient à colorer ce graphe en autant de couleurs que la grille horaire compte de plages, alors on a un horaire respectant les contraintes du modèle de base : il suffit d'associer à chaque cours la plage horaire qui a la même couleur.

Voyons comment on peut faire cette coloration selon le type de méthode utilisé.

4.1.2 Méthodes classiques

Avec une méthode classique, il suffit de construire le graphe défini à la section précédente et de lui appliquer la méthode, pour en obtenir une coloration plus ou moins optimale. Deux cas se présentent alors.

- La coloration utilise plus de couleurs qu'il n'y a de plages horaires.

Dans ce cas, cette coloration risque fort d'être inutilisable, et il vaut mieux recommencer, soit avec une autre méthode de coloration, soit avec un graphe plus simple, obtenu en enlevant des arêtes représentant des indisponibilités de professeurs qui en ont beaucoup. Peut-être la nouvelle coloration obtenue sera-t-elle meilleure.

Notons que s'il n'y a qu'une ou deux couleurs de trop, il est sans doute possible de former un horaire valable en remaniant ce résultat, puisque les méthodes de coloration, non optimales, utilisent très souvent un peu plus de couleurs que nécessaire.

- La coloration utilise juste autant de couleurs qu'il n'y a de plages horaires.

  Dans ce cas, on a un horaire vérifiant les contraintes posées. Mais il en existe certainement d'autres, que les méthodes de coloration, déterministes, ne donneront pas. Par conséquent, cet horaire "premier jet" a de grandes chances de s'améliorer si on le retravaille, pour tenir compte d'autres contraintes, de locaux, d'heures de fourches, etc.

Remarque : la coloration aura nécessairement besoin d'une couleur par plage horaire, puisque celles-ci sont représentées par des sommets tous reliés entre eux (formant un sous-graphe complet). Il n'y a donc pas d'autre cas.

4.1.3 Méthodes avec fonction objectif

Dans le chapitre précédent, nous avons donné une idée de la façon d'utiliser les méthodes d'optimisation avec fonction objectif en coloration de graphes. Nous allons ici revenir en détail sur ce point, en définissant les états du système, la fonction objectif, la façon de passer d'un état à un état voisin et enfin la façon de produire l'état initial.

Soit $G = (X, V)$ un graphe non-orienté de n sommets, à colorer en k couleurs.

Les **états** du système seront toutes les partitions de X en k ensembles $C_1, \ldots, C_k$, c'est-à-dire toutes les colorations impropres de G.

Définition :

Nous appellerons **arête monochrome** une arête dont les deux sommets ont la même couleur, c'est-à-dire ici dont les deux sommets appartiennent au même

ensemble.

La valeur de la **fonction objectif** en un état est le nombre d'arêtes monochromes existant dans cet état :

$$f_{Base} = \sum_{i=1}^{k} \# \left\{ \overline{(x_1, x_2)} \in V \ \dagger \ x_1, x_2 \in C_i \right\}.$$

La valeur minimale de cette fonction est 0 ; s'il n'existe pas d'arêtes monochromes dans une partition, c'est une coloration du graphe.

Les **transformations** sont les déplacements d'un sommet d'un ensemble à un autre (on conserve bien une partition de X). Par conséquent, deux états sont **voisins** *SSI* ils diffèrent par la couleur d'un et un seul sommet.

Notons que lors de la recherche d'un voisin, on se limite en fait aux transformations susceptibles d'améliorer la fonction objectif, c'est-à-dire celles qui éliminent une arête monochrome (quitte à en recréer d'autres). Pour choisir une transformation, on procédera donc comme suit :

- choisir un sommet d'une arête monochrome,
- choisir un ensemble-destination pour ce sommet, différent de son ensemble actuel.

Une transformation peut être décrite par un sommet, un ensemble origine et un ensemble-destination. Ce seront les trois composantes d'un élément de la liste des tabous, dans la méthode "Tabu-Search".

L'**état initial** pourra être produit de diverses manières.

La plus simple est la répartition aléatoire des sommets en k ensembles. Mais on peut gagner beaucoup de temps en utilisant une méthode plus "intelligente". Par exemple, si l'on connaît déjà une coloration en l couleurs, avec $l > k$, on peut l'exploiter en formant $k-1$ ensembles avec les sommets de $k-1$ des couleurs (par exemple celles groupant le plus de sommets) et en reprenant tous les autre sommets dans le $k^{\text{ème}}$ ensemble.

4.2 Extensions

4.2.1 Plage imposée

Le placement imposé peut être modélisé de façon similaire aux indisponi-

bilités : il suffit de relier, par autant d'arêtes,

- d'une part le sommet correspondant au cours contraint et
- d'autre part chacun des sommets correspondant aux plages, sauf celle qui est imposée.

Ceci peut s'appliquer aux deux types de méthodes.

4.2.2 Cours simultanés

On peut représenter les contraintes de simultanéité dans le graphe même, indépendamment de la méthode de coloration. Pour ce faire, on associe, à chaque groupe S de cours simultanés, un seul sommet x_S. Il sera relié aux autres en considérant qu'il correspond à un cours donné par tous les professeurs et suivi par toutes les classes concernés par les cours du groupe S. Ainsi, on trouvera une arête entre x_S et

- tout sommet correspondant à une plage horaire où un des professeurs est indisponible,
- tout sommet correspondant à un cours qui est donné par un des professeurs ou suivi par une des classes.

Lors de la lecture du résultat, on attribuera à chacune des heures de cours de chaque groupe la plage horaire (la couleur) donnée au sommet commun.

On peut également modéliser les cours simultanés via la fonction objectif. Celle-ci est alors une somme pondérée de la fonction f_{Base} utilisée précédemment et d'une fonction f_{Sim} définie comme suit :

$$f_{Sim} = \sum_{S \in \{\text{groupes de cours simultanés}\}} t_S,$$

où

$$t_S = \begin{cases} 0 & \text{si tous les cours du groupe } S \text{ ont la même couleur,} \\ 1 & \text{sinon.} \end{cases}$$

Cette fonction f_{Sim} compte le nombre de groupes de cours devant être simultanés, mais ne l'étant pas dans l'état courant.

Remarques :

1. Il faudrait discuter de deux "paramètres" de la fonction.
   Le premier, très important, est le poids relatif de f_{Base} et de f_{Sim} dans

f. On pourrait raisonnablement donner un poids élevé à f_{Sim}, du fait de sa faible valeur maximale (nombre de groupes de cours simultanés) par rapport à la valeur de f_{Base}. A défaut, les variations de f_{Sim} seront noyées dans celles de f_{Base}.
Le second serait une éventuelle pondération des différentes contraintes de simultanéité, via une valeur non-nulle des termes t_S qui dépendrait de S.

2. La valeur de la fonction devant diminuer petit à petit lorsque l'on s'approche de l'objectif, nous ne pouvons pas nous contenter d'une fonction booléenne valant 0 si toutes les contraintes de simultanéité sont vérifiées et 1 sinon.

   Par contre, on pourrait envisager de compter le nombre de paires de cours devant être simultanés mais ne l'étant pas. Le temps d'évaluation en serait accru. Mais cela introduirait une évolution plus progressive de la fonction. Le nombre d'itérations s'en trouverait vraisemblablement réduit, car l'algorithme serait "conscient" d'une amélioration lorsque, déplaçant un cours, il le rendrait simultané à quelques autres du même groupe. (Il faudrait implémenter les méthodes pour pouvoir peser le pour et le contre.)

3. La fonction objectif devra être reconstruite pour chaque problème. Elle n'est pas, comme celle du modèle de base, invariante à l'horaire à construire.

4.2.3 Cours consécutifs

Cette extension est nettement plus délicate à traiter.

Soient A et B deux heures de cours, B devant suivre A.

Une première idée qui vient à l'esprit est de représenter A et B par un seul sommet x_A, et d'interpréter le résultat en donnant à A la plage horaire (la couleur) attribuée à x_A et en donnant à B la plage suivante.

Mais on se heurte au problème de la représentation (par des arêtes) des interdictions entre B et un autre cours C donné par le même professeur ou suivi par la même classe. En effet, B n'est pas associé à un sommet propre, mais bien à un sommet x_A qui ne porte pas sa couleur. Par conséquent, une arête

placée entre x_A et un autre sommet x_C n'interdit pas au cours C correspondant à x_C de se dérouler en même temps que B.

Il semble que cette extension ne puisse pas être prise en charge dans un graphe. Pour en tenir compte, il faudrait modifier les heuristiques de coloration.

Par contre, si l'on utilise une fonction objectif, on peut procéder comme suit.

On conserve le graphe du problème de base, mais on ajoute à la fonction objectif un terme f_{Cons} comptant le nombre d'heures de cours devant être consécutives mais ne l'étant pas. Ceci demande de plus vastes connaissances pour cette fonction, puisqu'elle doit pouvoir dire si deux couleurs (deux plages horaires) données sont consécutives ou non, en tenant compte des passages d'un demi-jour à l'autre, interrompant la succession des plages.

Il faut remarquer que, comme pour la simultanéité, cette fonction n'est plus "standard", en ce sens qu'elle diffère d'un problème à l'autre. Elle dépendra en effet de la liste de cours consécutifs et de la grille horaire.

Enfin, comme dans le cas précédent, on pourra paramétrer la fonction objectif, d'une part par le poids relatif de ses deux composantes (f_{Base} et f_{Cons}), et d'autre part par l'importance relative des contraintes de consécutivité.

4.2.4 Heures de fourche

Le problème des heures de fourche est le plus difficile à formuler (même en français) de façon précise, comme nous l'avons vu plus haut. Peut-on dès lors l'exprimer en termes de sommets et d'arêtes dans un graphe ? Nous n'en avons pas trouvé le moyen. Par contre, lorsque l'on dispose d'une fonction objectif, ce problème d'optimisation semble bien plus abordable.

L'essentiel est de trouver un algorithme d'évaluation de la fonction f_F comptant le nombre total d'heures de fourche. On peut, par exemple, prendre le suivant.

- Pour chaque professeur,
  à partir de la liste des sommets associés aux heures de cours donnés par ce professeur (établie lors de la formation du graphe), établir la liste des couleurs associées à ces sommets, c'est-à-dire la liste des plages horaires où ce professeur donnerait cours si l'état courant était retenu ;

- Pour chaque professeur et pour chaque journée,

compter le nombre de plages libres séparant les première et dernière heures occupées de la journée ;

- Sommer le tout pour obtenir le nombre total d'heures de fourche.

Lors de la coloration, on utilisera comme fonction objectif une somme pondérée des fonctions f_{Base} du problème de base et f_F évaluée comme décrit ci-dessus. Ainsi,

$$f = P.f_{Base} + f_F,$$

où P est un coefficient que l'on fixera selon l'importance donnée au problème des heures de fourche. Celui-ci sera d'autant plus négligé que P est élevé.

4.2.5 Locaux

L'extension permettant l'attribution des locaux est réalisable avec une méthode de coloration de graphe avec fonction objectif. C'est cependant assez compliqué et prendra beaucoup de temps lors de l'exécution. Mais le problème est suffisamment important et difficile à résoudre en pratique pour qu'une résolution informatisée puisse être utile, même si elle est peu efficace en temps.

Examinons d'abord le cas des méthodes de coloration classiques, qui, à notre avis, ne permettent pas d'implémenter la localisation, qui devra par conséquent se faire à part. Nous verrons par après pourquoi l'usage d'une méthode à fonction objectif offre plus de possibilités.

4.2.5.1 Méthodes classiques

4.2.5.1.1 Localisation simultanée

Avec ces méthodes, il nous semble impossible d'attribuer simultanément les plages horaires et les locaux. Le problème est le suivant : nous avons vu que deux cours ne pouvaient avoir la même couleur s'ils ont le même professeur ou la même classe. Or ceux-ci leur sont attribués définitivement. Par contre, les locaux n'étant pas fixés, on ne peut évidemment pas exprimer, par une arête entre deux sommets, qu'une heure de cours ne peut avoir la même couleur qu'une autre qui se donne dans le même local. Et, inversement, les heures de cours n'étant pas non plus fixées a priori, on ne peut pas exprimer dans le

graphe que deux cours se donnant à la même heure ne peuvent recevoir le même local.

Serait-il possible de colorer sur le couple moment-local, chaque couleur correspondant non plus à une plage horaire, mais à une sorte de localisation dans l'"espace-temps", fixant en même temps le moment et l'endroit où se donne le cours ? C'est une mauvaise idée. Par exemple, deux cours A et B suivis par la même classe ne peuvent être donnés en même temps (mais peuvent l'être dans le même local). Si A reçoit une couleur, il faudrait, pour satisfaire les contraintes, interdire à B non seulement cette couleur, mais aussi l'ensemble de toutes celles qui correspondent à la même plage horaire. Soit, au total, autant de couleurs qu'il n'y a de locaux. C'est impossible en coloration de graphe classique, et une définition plus générale de la coloration nécessiterait certainement des algorithmes autrement plus complexes que ceux que nous utilisons !

A défaut de pouvoir déterminer *simultanément* le local et le moment où vont se donner les heures de cours, nous avons réfléchi aux possibilités de localisation *antérieure* ou *postérieure*, et à leur combinaison.

4.2.5.1.2 Localisation a priori

Dans ce cas, on attribue un local à chaque cours, puis on forme le graphe de base, et on y ajoute des arêtes supplémentaires, reliant les sommets correspondant aux cours se donnant dans le même local (pour interdire que ceux-ci se donnent en même temps). La coloration sera donc plus fortement contrainte. Cela pose le problème d'une répartition judicieuse et équilibrée des locaux, surtout si l'on n'en a que le strict nécessaire. A défaut d'une bonne localisation, on va (au mieux) réduire le nombre de colorations possibles, et par conséquent perdre des possibilités d'optimisation sur d'autres critères (comme les heures de fourche). Et au pire, on risque d'empêcher toute coloration en un nombre de couleurs acceptable.

4.2.5.1.3 Localisation a posteriori

Ici, on colore le graphe comme dans le modèle de base, et on s'efforce ensuite d'attribuer les locaux. Mais on risque de tomber sur une mauvaise coloration, pour laquelle aucune localisation n'est possible. Il faudrait donc que la méthode de coloration fournisse plusieurs solutions. Or les méthodes classiques sont déterministes. Par conséquent, à moins d'utiliser une méthode à fonction

objectif, il faudra en pratique remanier l'horaire produit, avant ou en parallèle à l'attribution des locaux.

Remarquons cependant que, si l'on dispose d'un bon horaire sans locaux, de quelque manière qu'il ait été construit, l'attribution des locaux a posteriori peut se faire automatiquement par une nouvelle coloration. Le graphe à colorer peut être défini comme suit :

Sommets :

- Associer un sommet à chaque local.
- Associer un sommet à chaque cours.

Arêtes :

- Relier par une arête chaque paire de sommets-*locaux*.

(Conséquence : chaque local aura une couleur différente.)

- Relier par une arête chaque paire de sommets correspondant à des cours donnés en même temps.

(Conséquence : deux cours donnés en même temps auront une couleur différente.)

- Relier par une arête un sommet-*local* et un sommet-*cours SSI* ce cours ne peut se donner dans ce local (par exemple parce qu'il est trop petit, ou qu'il ne contient pas le matériel adéquat).

(Conséquence : un cours n'aura pas la même couleur qu'un local où il ne peut avoir lieu.)

L'horaire avec locaux sera alors construit en associant à chaque cours le local qui a reçu la même couleur.

4.2.5.1.4 Localisation partielle a priori

Peut-être faut-il chercher la meilleure solution dans une combinaison des deux approches que nous venons d'examiner. Si l'on fixe, pour certains locaux bien choisis, une liste de cours qui devront s'y tenir, avant de colorer le graphe construit sur base de cette localisation partielle, l'horaire obtenu pourrait être plus facile à compléter manuellement. Le choix de ces locaux devra se baser sur le potentiel de difficultés qu'ils représentent. Par exemple, les locaux spécifiques (tels que salles de gymnastique, laboratoires de langues ou de sciences), et les locaux fort demandés pour leur grande capacité, devraient être attribués d'office à certains cours. Ceux-ci seraient alors placés automatiquement dans des plages horaires différentes, grâce aux arêtes reliant les cours se donnant dans un même local.

4.2.5.2 Méthodes avec fonction objectif

L'utilisation d'une fonction objectif ne change rien au fait que nous ne voyons pas comment représenter *dans le graphe* les contraintes d'attribution des locaux. Par contre, on peut envisager une modélisation via la fonction objectif. C'est cependant particulièrement complexe. Voici quelques pistes de résolution, mettant en lumière les difficultés qui pourront se poser.

4.2.5.2.1 Localisation simultanée

Pour effectuer une localisation simultanée, il faudra tout d'abord redéfinir les états du système. Par exemple, on pourrait utiliser deux partitions indépendantes. La première, comme précédemment, répartissant les sommets en ensembles de couleur[1] $C_1, \ldots, C_k$. La seconde répartissant les sommets-*cours* (et eux seulement) en ensembles de local[1] $L_1, \ldots, L_l$. Un état serait ainsi caractérisé par une paire de partitions.

La fonction objectif f pourrait être une somme de deux fonctions de l'état courant, l'une (f_{Base}) comptant les arêtes monochromes, l'autre (f_{Loc}) les heures de cours données en même temps dans le même local (situations que nous appellerons **collisions**). La valeur minimale d'une telle fonction f est zéro[2].

Une transformation entre deux états voisins serait le déplacement d'un sommet soit entre deux ensembles de couleur, soit (pour des sommets-*cours* uniquement) entre deux ensembles de local. Ces déplacements concerneraient soit un sommet d'une arête monochrome, soit un sommet d'une collision.

Enfin, la génération de l'état initial pourrait être faite de diverses manières (aléatoires ou intelligentes).

Quels seraient les principales difficultés à résoudre ?

D'abord, la fonction de comptage des collisions,

$$f_{Loc} = \sum_{i=1}^{k} \#\{(x,y) \dagger\ x,y \in C_i \text{ et } \exists\, j \in \{1,\ldots,l\} : x,y \in L_j\},$$

[1] Ce ne sont pas des ensembles contenant *des* couleurs (ou *des* locaux), mais bien des ensembles contenant les sommets correspondant à *une* couleur (ou à *un* local).

[2] Atteindre cette valeur est nécessaire et suffisant pour avoir un horaire respectant
- les contraintes du modèle de base et
- la contrainte de localisation différente des cours simultanés,

mais non celles de capacité des locaux.

n'est pas simple à évaluer rapidement.

Ensuite, il faudra déterminer, pour chaque transformation, si elle modifiera la partition en couleurs ou la partition en locaux. C'est un problème assez complexe.

Enfin, il faut distinguer, pour tout ce qui touche la répartition en locaux, les sommets-*cours* des sommets-*plages horaires*. Ce n'est pas un gros problème, mais cela complique un peu les choses.

4.2.5.2.2 Localisation a posteriori avec rétroparcours

D'autre part, et ceci est sans doute plus intéressant, on peut effectuer une attribution des locaux a posteriori, mais avec une sorte de rétroparcours si on rencontre une attribution des plages horaires qui ne permet pas de localisation. On utilisera pour cela le non-déterminisme des méthodes de coloration avec fonction objectif, par exemple de la façon suivante.

Une fois obtenue une coloration du graphe du modèle de base (et donc un horaire), on pourra réaliser à l'aide d'une autre coloration une attribution de locaux sur base de cet horaire. Si l'on n'y est pas arrivé après un nombre d'itérations fixé, certaines collisions restant dans la meilleure solution obtenue, on revient au premier algorithme en le forçant à abandonner la solution qu'il avait donnée. Ceci peut se faire en perturbant cette solution par quelques transformations[3] choisies d'après les collisions restantes. Par exemple, en déplaçant les sommets concernés par le plus grand nombre d'entre elles. Le nombre de transformations pourrait d'autre part dépendre du nombre total de collisions.

Si, après un certain nombre de boucles *coloration de base - coloration de localisation - perturbation - coloration de base - ...*, on n'est arrivé à aucun résultat, on abandonnera, en donnant les "moins mauvaises" solutions obtenues.

4.3 Conclusion

Les méthodes de coloration classiques offrent peu d'intérêt. Nous avons vu en effet que

[3] Si l'on travaille avec la méthode Tabu-Search, on pourra inclure ces transformations dans la liste des tabous, pour avoir une bonne garantie de ne pas retomber sur le même horaire.

- leur objectif de minimiser le nombre de couleurs n'est pas vraiment adapté aux problèmes d'horaires ;
- leur déterminisme ne leur permet pas de donner une variété de solutions qui permettrait une sélection par les "utilisateurs" (sur bases de critères informels ne pouvant par être pris en compte par un modèle) ;
- bien peu de contraintes peuvent être prises en charge, le modèle du graphe étant beaucoup trop rigide.

Peut-on espérer contrebalancer tous ces défauts par les grandes qualités de la programmation logique, en implémentant ces méthodes en Prolog et en les adaptant à ce langage ? On verra plus loin les résultats déplorables que nous avons obtenu dans nos tentatives en ce sens.

Les méthodes avec fonction objectif semblent par contre assez prometteuses. Les solutions que nous avons ébauchées dans ce chapitre devraient être creusées plus avant, mais cela ferait sans doute la matière d'un mémoire entier.

Une question en particulier est sujette à controverse, concernant la "philosophie" même de ces méthodes.

Nous avons proposé, par exemple pour traiter le cas des cours simultanés, de scinder la fonction objectif en deux composantes, l'une comptant les violations des contraintes de base, l'autre celles de la contrainte de simultanéité. Au-delà du problème de la pondération de ces contraintes se pose la question de leur influence respective sur l'évolution du système. Dans le modèle de base, les transformations s'attachent à éliminer les arêtes monochromes. Doivent-elles ici viser parfois les arêtes et parfois les cours séparés de leur groupe de simultanés ? Et comment faut-il faire ce choix ?

Il existe une autre possibilités, correspondant à une vision différente du principe des méthodes d'optimisation avec dégradation contrôlée. Ces méthodes se basent sur un hasard légèrement dirigé pour trouver l'optimum, et se passent de toute information autre que les valeurs locales de la fonction objectif. Pourquoi dès lors essayer de les guider en leur imposant (par exemple) l'élimination des arêtes monochromes ? La seule recherche de ces arêtes est en effet suffisamment difficile pour ralentir la recherche d'une transformation, par rapport à la sélection aléatoire d'un sommet et d'un ensemble de destination. Une optimisation intelligente gagnera en nombre d'itération[4], mais perdra sans doute bien

[4]Et encore, sans doute pas de manière très marquée avec Tabu-Search, car son système de

plus sur le temps de chaque itération.

sélection du meilleur voisin généré réduit l'écart entre les générations aléatoire et intelligente.

Chapitre 5

PROGRAMMATION EN PROLOG

5.1 Déroulement général

Nous avons d'abord implémenté en Prolog la méthode CSG de coloration de graphe. Quelques tests ayant montré la totale inefficacité de ce programme, nous avons programmé la méthode LF, toujours en Prolog, dans l'espoir que cette méthode bien plus simple donnerait de meilleurs résultats. Il le furent en effet, mais pas suffisamment[1].

A la lumière de cette tentative, il nous a semblé que faire de la programmation traditionnelle dans un langage dédié à la programmation logique n'est pas intéressant. Nous avons cependant voulu confirmer cette première impression.

Par la suite, nous avons donc, en parallèle :

- D'une part implémenté la recherche d'horaires dans une approche non-déterministe (plus proche de l'esprit de la programmation logique) et défini une syntaxe pour la description des problèmes d'horaire. Pour une meilleure structure des programmes, nous avons écrit un module des données (baptisé "mDO") renfermant toute l'information concernant cette syntaxe et un module des résultats (appelé "mRE") traitant seul la représentation de l'horaire.

- D'autre part complètement révisé l'implémentation de la méthode CSG et la représentation des graphes. Nous avons ainsi redéfini le type abstrait GRAPHE (module "taGR") et apporté quelques modifications au programme de la méthode LF, pour l'adapter aux nouvelles structures de données.

Comme on le verra, les résultats obtenus par l'approche non-déterministe sont fort encourageants. Quant aux nouvelles versions des méthodes de coloration, elles donnent de moins bons résultats que les premières.

[1]Environ 10 secondes CPU pour colorer un graphe aléatoire de 35 sommets et de 20 %de densité. (Voir résultats plus complets au paragraphe 5.2.2).

5.2 Coloration de graphe

Pour construire un horaire par coloration de graphe, il faut procéder en trois étapes :

- la lecture du fichier de description du problème et la formation du graphe correspondant ;
- la coloration de ce graphe ;
- l'interprétation de cette coloration en un horaire et l'écriture du fichier des résultats.

Dans les premières versions, la partie centrale, implémentée la première, ne nous a pas donné de résultats satisfaisants. Les deux autres parties, effectuant la connexion avec le problème des horaires, n'ont pas été réalisées. Brièvement, nous décrirons la représentation du graphe utilisée et les résultats des tests de coloration effectués.

Dans les secondes versions, nous avons implémenté les trois étapes (ce sont ces programmes complets que nous listons en annexe). Si la première fonctionne bien, la coloration donne des résultats pires encore que précédemment, malgré la révision de la structure des données destinés à les améliorer. Cette nouvelle structure sera cependant rapidement présentée ici, pour faciliter la compréhension des programmes.

Les algorithmes (CSG comme LF) sont très précis et directifs, limitant l'apport créatif du programmeur. Une description des fonctions écrites serait donc peu intéressante ; nous en laisserons le soin aux courtes spécifications données dans les listings.

5.2.1 Représentation du graphe : première version

La première idée que nous avions était d'utiliser une variable pour représenter le graphe manipulé, et de transmettre cette variable d'une procédure à l'autre, comme paramètre. Dans ce contexte, la meilleure solution nous a semblé être une structure de listes emboîtées, les listes étant bien adaptées à la structure récursive de Prolog.

Nous avons donc décidé de représenter un graphe par une liste de termes de la forme

```
Sommet --- Adjacents
```

où "Sommet" est le nom d'un sommet du graphe, "Adjacents" est la liste des noms des sommets adjacents à ce sommet, "---" est un opérateur infixé défini pour la cause.

Les graphes étaient traités par les fonctions suivantes, écrites dans un fichier de fonctions "d'usage général" :

`adjacents(Sommet,Graphe,Liste_Adjacents)`
(donne les adjacents d'un sommet)

`liste_sommets(Graphe,Liste_Sommets)`
(donne la liste des sommets)

`complet(Graphe)`
(réussit si le graphe est complet)

`sommet_deg_max(Graphe,Sommet_Degre_Max,Degre)`
(donne le sommet de degré maximal)

`former_sous_graphe(Graphe,Liste_Sommet_Generateurs,Sous_Graphe)`
(forme le sous-graphe engendré par les sommets d'une liste)

`assert_degres(Graphe,Code)` et `retract_degres(Code)`
(respectivement, ajoute et retire de la base de connaissance des clauses donnant le degré des sommets)

`somme_degres(Liste_Sommets,Code,Somme_Degres)`
(donne la somme des degrés des sommets d'une liste)

`sommet_deg2_max(Graphe,Sommet_Degre2_Max,Degre2)`
(donne le sommet de degré d'ordre deux maximal)

L'attribution d'une couleur à un sommet etait représentée, indépendamment du graphe, par une clause ajoutée à la base de connaissances, de la forme

```
couleur(Sommet,Couleur)
```

où "Sommet" est le nom d'un sommet, "Couleur" est le numéro de sa couleur.

5.2.2 Premiers tests

Pour tester les programmes de coloration, nous avons utilisé une série de 18 graphes aléatoires, en six groupes de trois. Les groupes de trois graphes sont les suivants :

- graphes de 15 sommets
  - graphes de 21 arêtes (densité de 20 %)
  - graphes de 42 arêtes (densité de 40 %)
- graphes de 25 sommets
  - graphes de 60 arêtes (densité de 20 %)
  - graphes de 120 arêtes (densité de 40 %)
- graphes de 35 sommets
  - graphes de 119 arêtes (densité de 20 %)
  - graphes de 238 arêtes (densité de 40 %)

Le tableau 1 donne les résultats de coloration de ces graphes par les premières versions de nos programmes CSG et LF, sur **Alma** (un VAX 11/750 de 2 Mb de mémoire centrale et 400 Mb de disque, tournant à environ 1 Mips sous UNIX BSD 4.2).

Comment expliquer qu'ils soient aussi mauvais (pour le programme CSG surtout, qui s'interrompt en cours de coloration de graphes de quelques dizaines de sommets, pour cause de dépassement de capacité de la pile globale) ?

La pile globale sert, en gros[1], à la représentation des termes structurés (non-atomiques). Or, dans notre premier programme CSG, le graphe (un très grand terme structuré) passe de procédure en procédure, lesquelles sont assez nombreuses (contrairement à celles de LF). Bien que cette explication ne nous satisfasse pas vraiment, nous avons mis en cause la représentation du graphe.

5.2.3 Représentation du graphe : seconde version

Prolog étant conçu pour manipuler des faits et des règles stockés dans une base de connaissances, il nous a semblé intéressant de représenter le graphe sous forme de clauses dans la base. D'autre part, pour ne pas devoir corriger tout

[1] On en trouvera une description plus précise, mais assez technique, dans [2].

le programme en cas de nouveau changement, nous avons décidé de traiter le graphe comme un type abstrait. Nous avons de même isolé en un module les fonctions de recherche d'une clique (pour CSG), qui utilisent un sous-graphe. Nous décrivons dans les sections suivantes les fonctions de ces modules.

5.2.3.1 Module "taGR"

Le module "taGR" implémente un type abstrait GRAPHE, cachant à l'extérieur toute l'information concernant la représentation effective du graphe et de sa coloration, laquelle peut ainsi être revue sans que cela implique des modifications hors de taGR.

Ce module étant spécifiquement destiné à nos programmes de coloration, lesquels ne manipulent qu'un graphe, nous l'avons limité au traitement d'un seul graphe. Notons cependant qu'il est possible, sans difficultés, de généraliser ce module pour lui permettre de gérer plusieurs graphes à la fois, au prix toutefois d'une utilisation un peu moins agréable (du fait de l'ajout, dans chaque fonction, d'un paramètre servant à désigner un graphe) et d'une efficacité légèrement réduite.

Interface :

Le module taGR fournit les fonctions données ci-dessous, dont le nom, suffixé par les caractères "`_taGR`", indique plus ou moins bien le rôle. Pour plus de détails, on pourra se référer aux listings commentés donnés en annexe.

Opérateurs de construction du graphe (constructeurs)

```
init_graphe_taGR
ajout_sommet_taGR(Sommet)
signaler_groupe_taGR(Groupe,Liste_Cours)
def_adjacents_taGR(Sommet,Liste_Adjacents)
attribuer_couleur_taGR(Sommet,Couleur)
```

Opérateurs de saisie de données (sélecteurs)

```
ens_sommets_taGR(Ens_Sommets)
adjacents_taGR(Sommet,Ens_Adjacents)
couleur_attribuee_taGR(Sommet,Couleur)
couleurs_interdites_taGR(Sommet,Ens_Couleurs)
```

`coloration_taGR(Coloration)`

(Remarque : "`Coloration`" est une liste de termes de la forme "`couleur(Sommet,Couleur)`", où "`Sommet`" est un sommet et "`Couleur`" sa couleur.)

`groupe_taGR(Groupe,Liste_Cours)`

`coherence_taGR`

Représentation :

Le graphe et sa coloration sont représentés par des clauses introduites dans la base de connaissances. Ces clauses, de la forme

`sommet_taGR(Sommet,Ens_Adjacents,Couleur).` ,

asssocient un sommet, l'ensemble de ses adjacents et sa couleur.

Le premier argument est le nom du sommet.

Le deuxième est

- soit l'atome "`inc`", si les adjacents du sommet sont encore **inc**onnus (ils doivent être définis à l'aide de l'opérateur `def_adjacents_taGR`) ;
- soit la liste, triée et sans doubles, des adjacents du sommet.

Le dernier argument est

- soit l'atome "`inc`" si les adjacents sont inconnus, ce qui interdit toutes manipulation au niveau des couleurs ;
- soit une liste à un seul élément, qui est la liste (triée et sans doubles) des couleurs interdites au sommet, c'est-à-dire des couleurs présentes parmi ses adjacents, si le sommet n'est pas encore coloré ;
- soit le numéro de la couleur attribuée au sommet, s'il est coloré.

Cette représentation utilise une liste de liste dans le seul but d'accéder plus aisément aux données en différenciant mieux les trois cas (grâce au processus de "*matching*" de Prolog, vérifier qu'un terme est une liste à un élément est plus facile que vérifier qu'un terme est une liste quelconque).

Remarque : l'usage de listes triées et sans doubles permet une plus grande efficacité des manipulations "ensemblistes" (recherche d'un élément, union, intersection, différence, etc.) .

5.2.3.2 Module de recherche d'une clique

Pour la méthode CSG, nous avons également isolé dans un fichier les fonctions relatives à la recherche d'une clique. Ces fonctions n'ont pas de suffixe particulier. Pour composer la clique (*cf* algorithme en section 3.1.2.3.2), elles utilisent un sous-graphe du graphe à colorer, qui est réduit au fur et à mesure du travail.

Fonctions :

Les fonctions de recherche de clique sont en deux groupes : celles qui manipulent le sous-graphe, et celles qui composent la clique. Les premières sont écrites de façon à cacher la représentation du sous-graphe (comme pour un type abstrait), mais sont spécialement destinées à la recherche de clique, et n'ont donc pas une utilité assez générale pour constituer un "type abstrait".

Fonctions de manipulation du sous-graphe

`creer_sous_graphe(Ens_Sommets_Generateurs)`
(remarque : cette procédure construit un sous-graphe, engendré par les sommets de `Ens_Sommets_Generateurs`, du graphe traité par le module taGR)
`reduire_sous_graphe(Ens_Sommets_Generateurs)`
`sg_est_vide`
`adjacents_dans_sg(Sommet,Ens_Adj)`
`sommet_degre_max_dans_sg(Sommet)`

Fonctions de recherche d'une clique

`constr_clique(Clique_a_agrandir,Clique)`
`recherche_clique(Clique)`

La fonction utile pour le programme de coloration CSG est `recherche_clique/1`.

Représentation :

Le sous-graphe est représenté par des clauses insérées dans la base de connaissance, de la forme

"`sommet_sg(Sommet,Ens_Adjacents).`",

où "`Sommet`" est le nom d'un sommet
et "`Ens_Adjacents`" est l'ensemble (liste triée sans doubles) des adjacents de ce

sommet dans le sous-graphe.

5.2.4 Tests des secondes versions

Contrairement à nos attentes, les secondes versions de nos programmes de coloration CSG et LF sont moins bonnes encore que les premières. Pour colorer les mêmes graphes de tests, la méthode LF prend environ deux fois plus de temps, et la méthode CSG trois fois plus.

Comment l'expliquer ? Faut-il croire que l'usage de fonctions réalisant une interface entre les données et le programme ralentit l'exécution ?

Cette explication ne nous convainc vraiment pas, comme c'était le cas pour les premiers programmes.

5.2.5 Conclusions

La principale conclusion que nous tirons de cette expérience est la totale inadéquation de la programmation logique à des méthodes déterministes comme celles que nous avons implémentées. D'autant que, comme on le verra dans la section suivante, nous obtenons de bons résultats avec une méthode non-déterministe, parcourant tout l'espace des solutions.

5.3 Programmation logique

5.3.1 Introduction

Dans les pages qui suivent, nous allons décrire la partie la plus tangible de notre travail. Nous avons implémenté, en Prolog, un algorithme de construction d'horaires par exploration systématique de l'espace des solutions[1]. Cet algorithme, qui nous semble respecter l'esprit de la programmation logique, donne de très bons résultats. Il présente l'avantage de pouvoir donner en une seule exécution toutes les solutions du problème. Mais il s'agit d'un programme "expérimental", qui prétend seulement vérifier s'il est possible de construire des horaires en Prolog : l'ergonomie est passée au second plan.

Pour commencer, nous décrivons les deux modules réalisant l'interface du "programme principal" avec d'une part le fichier des données et la représentation du problème d'horaire à résoudre, et d'autre part le fichier des résultats et la représentation de l'horaire construit.

Ensuite, nous expliquons la méthode de recherche employée, et la façon de prendre en charge les différentes contraintes et extensions fonctionnelles qui nous intéressent.

On trouvera en annexe les listings de toutes ces fonctions, ainsi que celui du petit fichier de commandes qui consulte tous les autres.

5.3.2 Module des données

Le module des données (baptisé "mDO") réalise l'interface entre les données et le programme de construction d'horaire, de sorte que celui-ci puisse tout ignorer de la syntaxe de description du problème.

5.3.2.1 Fonctions de l'interface

mDO offre aux modules qui l'utilisent une série de fonctions, dont la liste est donnée ci-dessous. Leurs noms sont suffixés des caractères "`_mDO`" pour les identifier plus aisément dans les programmes. Une courte spécification en est donnée dans les fichiers listés en annexe.

[1]Nous en parlerons parfois en les termes "programme non-déterministe".

Fonctions mDO :

Fonctions destinées au traitement du problème

`lire_donnees_mDO(Nom_Fichier_Donnees)`
`plage_horaire_mDO(Plage)`
`liste_cours_mDO(Liste_Cours,Avec_Tri)`
(voir remarque ci-dessous)
`cours_plage_incompat_mDO(Cours,Plage)`
(un cours et une plage sont incompatibles *SSI* un professeur qui donne ce cours est indisponible pour cette plage)
`cours_independants_mDO(Cours1,Cours2)`
(deux cours sont indépendants *SSI* les professeurs qui les donnent et les classes qui les suivent sont différents)
`groupe_de_simultanes_mDO(Cours,Nom_Groupe)`
`plages_consec_mDO(Premiere_Plage,Deuxieme_Plage)`
`cours_consec_mDO(Premier_Cours,Deuxieme_Cours)`

Fonctions nécessaires à l'écriture des résultats

`caract_plage_mDO(Plage,Description_Plage)`
`caract_cours_mDO(Cours,Description_Cours,Liste_Profs,Liste_Classes)`
`caract_classe_mDO(Classe,Description_Classe)`
`caract_prof_mDO(Prof,Description_Prof)`
`caract_local_mDO(Local,Description_Local)`

Remarque importante :

La fonction `liste_cours_mDO` propose un tri des cours, optionnel. Ce tri se fait par ordre de liberté croissante, donnant ainsi en premier les cours susceptibles d'être les plus difficiles à placer dans la grille. La "liberté" d'un cours est estimée selon la formule suivante :

nombre de plages libres (selon les disponibilités professorales)
- nombre de cours simultanés au cours
- 2 * nombre de contraintes de consécutivité affectant le cours.

Cette formule assez arbitraire (qui n'a fait l'objet d'aucune analyse d'optimalité) donne de bons résultats sans coûter très cher en temps de calcul. Nos tests ont montré que ce tri permettait d'éviter au programme d'énormes pertes

de temps, provoquées simplement par une déclaration des cours dans un ordre défavorable.

5.3.2.2 Syntaxe de description du problème

Pour définir la syntaxe, nous avons d'abord pris comme principe que la description du problème devait ressembler à du français. Par ailleurs, pour que ces phrases soient utilisables sans manipulation par un programme Prolog, il faut qu'elles aient la structure de clauses du langage, et même, idéalement, de faits.

Pour satisfaire cette seconde exigence, nous avons opté pour l'utilisation de faits avec un foncteur et des arguments. Le foncteur décrit la nature d'une information sur le problème et les arguments donnent les objets concernés par cette information. Mais ce foncteur et ces arguments doivent être des atomes Prolog, c'est-à-dire principalement commencer par une lettre minuscule et être écrits d'un seul tenant[2] (à défaut, ils doivent être encadrés d'apostrophes).

Pour répondre à la première condition, nous devons utiliser une structure assimilable à la structure française sujet-verbe-complément. Nous avons choisi une forme objet-relation-objet, la relation étant exprimée en utilisant (notamment) un verbe, et les objets jouant les rôles (respectivement) de sujet et de complément. Par exemple, pour exprimer qu'un cours C est donné par un professeur P, nous voulons pouvoir utiliser tout simplement une phrase comme "*C est donné par P*", en assimilant "*C*" au sujet, "*P*" au complément et "*est donné par*" au verbe.

Selon les concepts Prolog, cela veut dire que le foncteur ("verbe") doit accepter deux arguments, écrits l'un devant lui et l'autre derrière. Ceci a nécessité la définition des foncteurs comme des opérateurs de type infixé[3].

D'autre part, pour pouvoir utiliser commodément des atomes Prolog comme arguments, il nous a semblé intéressant d'introduire, pour chaque "objet" ma-

---

[2]En gros, un atome Prolog peut contenir des lettres, des chiffres et le caractère de soulignement (ou "*underscore*"), lequel sert généralement à relier des mots "normaux" pour en faire un atome.

[3]Prolog permet en effet au programmeur de définir ses propres opérateurs, en en donnant la précédence, la règle d'associativité, le type (préfixé, infixé, postfixé) et le nom. Ces opérateurs auront nécessairement un ou deux arguments.

nipulé, une association nom-description, le nom étant un atome Prolog assez court[4] et la description[5] étant un texte de longueur et de composition quelconques, encadré des apostrophes nécessaires à en faire un atome.

Un problème se pose dans le cas des contraintes de simultanéité, celles-ci pouvant affecter les cours par groupes (alors que la plupart des contraintes relient les objets deux par deux).

Considérons un groupe de N cours dont on veut la simultanéité. Une possibilité est d'écrire une chaîne de $N-1$ relations, chacune reliant deux cours du groupe. Une autre est de donner $N.(N-1)/2$ relations, une pour chaque paire de cours du groupe. Ces deux solutions sont peu satisfaisantes. La première parce que, outre le traitement de la transitivité qu'elle impose, elle nous semble produire une description manquant de clarté pour un lecteur humain. La seconde, car elle est fastidieuse à écrire dans le cas de groupes d'une certaine taille.

Nous avons préféré une association de chaque cours avec son groupe, soit N clauses. Le groupe ne doit pas faire l'objet d'une description, son nom suffit (il ne doit en effet pas apparaître dans le fichier des résultats).

Nous avons ainsi abouti à la syntaxe donnée plus bas, pour décrire un problème d'horaire (modèle de base étendu aux contraintes de placement imposé, de simultanéité et de consécutivité et à l'attribution des locaux).

Remarques :

1. Prolog exige que chaque clause se termine par un "full stop", c'est-à-dire un point suivi d'un blanc ou d'un passage à la ligne (lequel améliore la lisibilité).

2. Comme la plupart des langages de programmation, le Prolog que nous utilisons n'accepte pas les caractères accentués ni les cédilles.

---

[4] Ce nom est une abréviation permettant une écriture plus rapide et facile du fichier de données. Il doit bien entendu être unique.

[5] Cette description est le nom complet de l'objet, destinée à apparaître dans le fichier des résultats.

Syntaxe :

- Descriptions des "objets" manipulés : format

```
<nom de l'objet> <operateur> <description de l'objet>.
```

- plages horaires : opérateur `est_la_plage` ;
- cours : opérateur `est_le_cours` ;
- professeurs : opérateur `est_le_prof` ;
- classes : opérateur `est_la_classe` ;

Exemples :

```
mardi_3e  est_la_plage  'mardi 3eme heure (10h40-11h30)'.
math_4B_3  est_le_cours  'Mathematiques - 4eme annee B (heure 3)'.
dupont  est_le_prof  'Jacques Dupont'.
c4B  est_la_classe  '4eme annee B'.
```

- Descriptions du problème et de ses contraintes : format

```
<nom d'un objet> <op\'{e}rateur> <nom d'un objet>.
```

- relation cours-professeur : opérateur `est_donne_par` ;
- relation cours-classe : opérateur `est_suivi_par` ;
- relation de consécutivité plage-plage : opérateur `est_consecutive_a` ;
- contrainte d'indisponibilité : opérateur `est_indisponible_pour` ;
- contrainte de simultanéité : opérateur `est_dans_le_groupe` ;
- contrainte de consécutivité : opérateur `doit_etre_consecutif_a` ;

Exemples :

```
math_4B_3  est_donne_par  dupont.
math_4B_3  est_suivi_par  c4B.
mardi_4e  est_consecutive_a  mardi_3e.
dupont  est_indisponible_pour  mardi_3e.
math_4B_3  est_dans_le_groupe  gr_simult_2.
math_4B_4  doit_etre_consecutive_a  math_4B_3.
```

5.3.3 Module des résultats

Le module des résultats (appelé "mRE") réalise l'interface entre le programme de construction d'horaire et la représentation de l'horaire construit[6]. Il contient aussi les fonctions d'écriture du fichier des résultats.

5.3.3.1 Fonctions de l'interface

Le module mRE implémente une série de fonctions, dont la liste est donnée ci-dessous. Les noms des fonctions manipulant l'horaire construit sont suffixés des caractères "_mRE", ce qui permet de les identifier plus aisément dans les programmes. Nous considérons ces noms comme suffisamment parlants pour ne pas décrire ici toutes ces fonctions.

Une courte spécification est donnée dans les fichiers listés en annexe.

Fonctions mRE :

Fonctions de manipulation de l'horaire construit

```
init_grille_mRE
attribuer_plage_mRE(Cours,Plage)
attribuer_local_mRE(Cours,Local)
plage_attribuee_mRE(Cours,Plage)
local_attribue_mRE(Cours,Local)
```

Fonctions d'écriture du fichier de résultats

```
ecrire_entete(Tps_Init,Nom_Fich_Donn,Avec_Locaux,Avec_Tri,
Tps_Min_Recherche)
ecrire_solution(Temps_Ecoule,Avec_Locaux)
```

(écrit la solution qui vient d'être trouvée)

5.3.3.2 Représentation de l'horaire

L'horaire de cours est représenté par des faits enregistrés dans la base de connaissances. Ces faits indiquent les associations cours-plage et cours-local. Ce sont, respectivement,

[6] Ceci permettrait, le cas échéant, de modifier cette représentation sans devoir récrire d'autres fonctions que celles de mRE.

```
horaire(Cours,Plage).
```

et

```
localisation(Cours,Local).
```

5.3.4 Fonction principales

5.3.4.1 Modèle de base

Pour résoudre le problème d'horaire de base avec un programme logique, nous utilisons, pour l'essentiel, l'objectif :

- construire la liste des cours à placer dans l'horaire,
- planifier cette liste de cours (les placer dans l'horaire).

Pour planifier une liste de cours vide, il ne faut rien faire.
Pour planifier une liste de cours non vide, il faut :

1. trouver une plage horaire,
2. vérifier si cette plage convient au premier cours de la liste,
3. associer cette plage à ce cours dans la base de connaissances,
4. planifier le reste de la liste de cours.

Déroulement de cette procédure :

Le premier de ces quatre buts fournit une plage, prise "au hasard" (elles sont en fait données suivant leur ordre d'apparition dans la base de connaissances).

Si la plage convient (en fonction des associations cours-plage déjà effectuées), le deuxième but réussit, et l'association cours-plage est ajoutée à la base de connaissances par le troisième but. On peut alors essayer de planifier le reste (en fonction, toujours, de ce qui a déjà été fait). Au cas où ce serait impossible, les choix effectués jusque là n'étant pas bons, le quatrième but échouera, ramenant l'exécution au but 3 (rétroparcours). Celui-ci doit être programmé pour défaire au second passage l'association cours-plage qu'il avait établie au premier, puis pour échouer, transmettant le rétroparcours aux buts 2 (déterministe) puis 1.

Si une plage ne convient pas au cours que l'on cherche à placer, le deuxième but échoue, provoquant, par rétroparcours, la remise en question du premier : une autre plage est choisie, et on recommence.

Si aucune plage ne convient, toutes les "propositions" du premier but sont successivement refusées par le second. Cherchant une plage de plus, mais n'en trouvant pas, le premier but échoue, ce qui provoque l'échec, au niveau précédent, du quatrième but.

Pour vérifier qu'une plage convient à un cours C, il faut :

1. vérifier que tous les professeurs qui donnent le cours C sont disponibles pour cette plage horaire,

2. vérifier que les cours déjà placés dans l'horaire et associés à cette plage sont indépendants du cours C (classes et professeurs différents).

En Prolog, tout cela pourrait s'écrire :

```
creation_horaire :-
   liste_des_cours(LCours),
   planifier(LCours).


planifier([]).
planifier([Cours|Reste]) :-
   plage_horaire(Plage),
   association_possible(Cours,Plage),
   associer(Cours,Plage),
   planifier(Reste).


association_possible(Cours,Plage) :-
   professeurs_disponibles(Cours,Plage),
   liste_des_cours_en_plage(Plage,LCours_en_P),
   independance(Cours,LCours_en_P).
```

Le reste dépend surtout de la façon de représenter les connaissances, de la saisie des données et de la sortie des résultats. Nous avons isolé ces informations dans des fichiers différents, de façon à pouvoir en changer sans devoir modifier les fonctions principales. Ainsi, comme on l'a vu plus haut, les données sont traitées par le module mDO, la représentation de l'horaire et la sortie des résultats par le module mRE.

(Les listings que l'on trouvera en annexe montreront comment nous avons procédé exactement.)

5.3.4.2 Extension aux contraintes

5.3.4.2.1 Principe général

Pour prendre en charge les contraintes, deux approches sont possibles. La première consiste à vérifier, *après* avoir pris une décision, qu'elle les respecte. La

seconde consiste à les utiliser pour réduire, *avant* d'effectuer le choix, l'espace des possibilités. Selon la nature des contraintes, on accordera la préférence à l'une ou à l'autre technique, ou à une combinaison des deux.

Si l'on utilise des contraintes dont l'effet est d'interdire certains choix, en laissant plusieurs possibilités, il est plus intéressant de vérifier a posteriori que le choix est valable, et de le remettre en question sinon. En effet, l'autre méthode demanderait d'examiner toutes les possibilités, pour éliminer celles qui ne satisfont pas les contraintes, puis de choisir parmi celles qui restent. Autrement dit, il faudrait faire plus de travail qu'avec la première approche.

Par contre, pour des contraintes *d'obligation stricte*, c'est-à-dire des contraintes qui imposent un et un seul choix, la deuxième technique est à conseiller. Plutôt que de vérifier si une décision prise est bien la seule que l'on pouvait prendre, mieux vaut demander à l'avance quelle décision on doit prendre.

Le modèle de base n'utilise que des contraintes du premier type. Par contre, les extensions que nous considérons ici sont toutes du second type. Nous avons donc opté pour la seconde approche pour implémenter leur prise en charge. Après avoir déterminé si une décision était imposée (sinon, nous prenons un choix libre), nous vérifions qu'elle satisfait les contraintes du modèle de base.

Notons que nous sommes confrontés à un risque d'impossibilité bien plus important ici, les contraintes étant plus strictes.

Ainsi, si deux contraintes imposent deux décisions différentes, ou si une contrainte ne peut être satisfaite[7], le choix ne peut se faire, ce qui remet en question les décisions précédentes (par rétroparcours).

Par ailleurs, si une décision imposée s'avérait mauvaise par la suite (c'est-à-dire si, pour les décisions restant à prendre, aucune possibilité n'était valable), elle ne pourrait être remise en question, puisque c'était la seule possible. Ce serait alors la décision précédente qui serait considérée comme mauvaise.

Nous examinons ci-dessous les situations particulières engendrées par les trois types de contraintes que nous avons retenus.

5.3.4.2.2 Situations particulières

[7]Nous pensons ici aux contraintes de consécutivité. Si un cours B doit suivre un cours A, et si A est déjà placé en fin de journée, alors la contrainte ne peut être satisfaite : il n'existe pas de plage consécutive à celle de A.

5.3.4.2.2.1 Placement imposé

Le cas où une plage est imposée à un cours ne pose pas de difficultés particulières : la contrainte est toujours d'application et la recherche de la décision imposée ne présente aucune difficulté.

5.3.4.2.2.2 Simultanéité

Pour les contraintes de simultanéité de cours, un choix de plage n'est effectivement imposé que si un des cours du groupe de simultanés est déjà placé dans la grille. Si ce n'est pas le cas, nous considérons que la contrainte ne s'applique pas (ce qui est bien dans l'optique exploratoire consistant à expérimenter toutes les possibilités plutôt qu'à essayer de faire des choix intelligents, lesquels sont bien plus lents).

Remarque :

Dans la structure de données retenue, les cours simultanés sont déclarés comme appartenant à un "groupe", mais aucune vérification n'est faite de l'appartenance d'un cours à *au plus* un de ces groupes. Par conséquent, le programme doit pouvoir traiter le cas où un cours appartient à plusieurs groupes (dont les cours seront alors tous simultanés).

Considérons par exemple un cours B déclaré dans deux groupes de cours simultanés : d'une part, le cours B et un cours A, et d'autre part le cours B et un cours C. De plus, supposons que B apparaisse *après* A et C dans la liste des cours, de sorte qu'il sera placé après eux dans la grille. Alors, tant que A et C ne se verront pas attribuer "par hasard" la même plage horaire, les contraintes ne pourront pas être vérifiées par B, qui doit être simultané aux deux à la fois. Après bien des choix inutiles, le programme finira par trouver une solution, mais sa recherche aura été peu efficace.

Il est donc important de décrire soigneusement le problème à résoudre, mais pour des raisons d'efficacité seulement. Des données bâclées causeront une perte de temps, éventuellement considérable, mais pas un échec.

5.3.4.2.2.3 Consécutivité

Une contrainte de consécutivité ne s'applique que si un des deux cours concernés est déjà placé dans la grille.

Pour commencer, on regardera donc si le cours B à placer doit suivre un autre cours A, puis si ce cours A est déjà placé dans la grille, à une plage P. Si

c'est le cas, il y a contrainte. On détermine alors la plage imposée : c'est celle qui suit la plage P, **si elle existe**. Sinon, la contrainte ne peut être satisfaite, et il y a échec, remettant en question la décision précédente.

On procédera ensuite de façon similaire pour traiter le cas inverse, où le cours à placer doit *précéder* un autre cours.

Remarque :

Un cours peut être l'objet de plusieurs contraintes de consécutivité. Comme pour la simultanéité, notre programme les prend en charge individuellement, sans "intelligence".

Considérons par exemple la situation suivante : un cours B est consécutif à deux cours A_1 et A_2, mais ceux-ci ne sont pas déclarés simultanés. Supposons en outre que, dans la liste des cours, B vient *après* A_1 et A_2 : B ne sera placé dans la grille qu'après A_1 et A_2. Dans ce cas, lors de la décision pour B, les contraintes ne seront satisfaisables (et B ne pourra être placé) que si A_1 et A_2 ont, "par hasard", reçu la même plage horaire. Ceci provoque un grand nombre de tâtonnements, qui auraient été évités par la déclaration de A_1 et A_2 comme simultanés, mais n'empêche nullement le programme de trouver la solution. Cela prendra simplement plus de temps.

On voit ici encore l'importance d'une description bien réfléchie du problème.

5.3.4.2.3 Réalisation pratique

Notre programme utilise le système suivant, implémenté dans la procédure `choisir_plage/2` effectuant le choix d'une plage pour un cours donné. Nous réunissons dans un ensemble toutes les plages imposées à ce cours par une de nos contraintes.

Nous utilisons pour cela la primitive `setof/3` de Prolog, collectant dans un ensemble[8] tous les objets vérifiant une condition donnée, et échouant s'il n'existe pas de tels objets. Dans ce dernier cas, l'exécution bascule sur une autre clause de la procédure `choisir_plage/2`, qui choisit une plage "au hasard", exactement comme pour le modèle de base. Notons qu'un *cut* (!) placé juste après le `setof` empêche le recours à cette seconde clause en cas de succès de `setof`, c'est-à-dire si une contrainte s'applique pour imposer une plage horaire au cours traité.

Si l'ensemble des plages imposées contient plusieurs éléments, il est impossible de vérifier toutes les contraintes en même temps, et la procédure

[8]liste triée sans doubles.

`choisir_plage/2` échoue directement. Si l'ensemble contient une seule plage, celle-ci est imposée sans possibilité d'en prendre une autre. Le cas particulier des contraintes qui, comme la consécutivité, peuvent être d'application mais être impossible à satisfaire est traité en plaçant dans l'ensemble des plages imposées l'atome `impossibilite`. Si l'ensemble ne contient qu'un élément, mais qu'il s'agit de cet atome, alors la procédure échoue exactement comme s'il y avait plusieurs plages imposées.

Cette méthode présente l'avantage d'être assez facilement extensible à d'autres types de contraintes *d'obligation stricte* (qui, lorsqu'elles s'appliquent, ont pour effet d'imposer à un cours une et une seule plage). Il suffit en effet de pouvoir écrire une procédure qui fournit, par rétroparcours, chacune des plages imposées[9] à un cours du fait d'une contrainte du type en question. Cette procédure sera alors invoquée dans la condition que doivent vérifier les objets collectés par le `setof`.

5.3.4.2.4 Généralisation

La généralisation à des contraintes *d'obligation large* ou *d'interdiction* (c'est-à-dire restreignant le choix à une liste de possibilités, en imposant celles-ci ou en interdisant les autres) se fera aisément selon le principe utilisé pour le modèle de base, dont les contraintes sont de cette nature. Il suffira à cet effet de modifier la procédure vérifiant qu'une association cours-plage est possible.

5.3.4.3 Extension à l'attribution des locaux

La localisation peut assez facilement se greffer sur un programme Prolog effectuant la seule attribution des plages. Le principe général diffère peu. Ici, la planification d'une liste de cours consistera en :

1. planifier dans le temps (plages horaires) le premier cours de la liste,
2. planifier dans l'espace (locaux) ce cours, connaissant la plage où il a été placé,
3. planifier le reste de la liste.

---

[9]Un cours pouvant être soumis à plusieurs contraintes du même type, chacune lui imposant une plage, cette procédure doit pouvoir donner *toutes* les plages imposées, fussent-elles différentes (auquel cas il y aura échec de la procédure `choisir_plage`).

Le premier but se résout de la même façon que dans le cadre de la seule attribution des plages (*cf* section 5.3.4.1 et 5.3.4.2).

Le second but suit un principe similaire :

1. choisir un local,
2. vérifier que ce local est libre au moment où le cours se donnera,
3. vérifier que ce local convient au cours (du point de vue de sa capacité),
4. ajouter dans la base de connaissances l'association cours-local.

Exactement de la même façon que pour les plages, la localisation pourra faire l'objet de contraintes, qui seront prises en charge soit par la procédure choisissant un local, soit par celle vérifiant qu'il convient au cours.

(Nous avons donné quelques exemples de contraintes au chapitre 2.)

5.3.4.4 Heures de fourche : remarques

La programmation logique, avec son approche consistant à parcourir tout l'espace de résolution, n'est pas du tout adaptée aux problèmes d'optimisation de fonctions. Or, comme nous l'avons vu au chapitre 2, le problème des heures de fourche est un problème de minimisation d'une fonction entière (dont la définition varie avec l'objectif exact que l'on poursuit).

Pour donner la solution optimale, il faudrait donc essayer toutes les combinaisons, tous les horaires possibles. C'est inacceptable en pratique, le temps d'attente étant dans ce cas énorme.

Par contre, à titre d'aide, et à condition de pouvoir évaluer la fonction à minimiser (ce qui nécessite quelques informations supplémentaires, notamment pour connaître la structuration des plages en journées), on pourra donner avec chaque solution trouvée sa "valeur" pour ce critère, ou même une liste des heures de fourche de chaque professeur.

On peut cependant faire bien mieux, en abordant le problème d'une toute autre façon. On peut par exemple transformer l'objectif de minimisation en une contrainte de borne. Cette contrainte, à traiter selon les techniques exposées plus haut, limiterait le nombre d'heures de fourche, soit globalement, soit par professeur. La borne pourrait servir de paramètre de "mise au point" de l'horaire, paramètre réévalué à la hausse ou à la baisse selon qu'il y a pénurie ou abondance de solutions.

5.3.5 Tests

Nous avons testé l'efficacité de ce programme non-déterministe sur deux problèmes.

Le premier est un horaire "d'étude", de très petite taille : cinq plages horaires, trois locaux. Il compte sept solutions sans localisation, et deux solutions si l'on tient compte des locaux. Il contient tous les types de contraintes que nous avons implémenté[10].

Le second problème est réel : il s'agit d'une partie de l'horaire de l'institut d'informatique (FNDP-Namur), sur vingt plages et huit locaux.

Nous traitons brièvement ci-dessous des temps d'exécution et des occupations de mémoire.

Tous les tests décrits ici ont été effectués dans les conditions optimales (un seul utilisateur) sur **Alma**, un VAX 11/750 de 2 Mb de mémoire et 400 Mb de disque, tournant à environ 1 Mips sous UNIX BSD 4.2.

5.3.5.1 Problème d'étude

La description du problème d'étude ("horaire 0") est donnée en annexe, avec le fichier-résultat de la recherche avec localisation (deux solutions).

Nous avons comparé, sur une exécution complète (c'est-à-dire en laissant le programme essayer toutes les possibilités), trois objectifs :

- la construction de l'horaire sans localisation, sur base d'une description du problème amputée de tout ce qui concerne les locaux ;
- la construction de l'horaire sans localisation, sur base de la description complète du problème ;
- la construction de l'horaire avec localisation.

Le tableau 2 donne, en secondes CPU, les temps

- de consultation (c'est-à-dire de chargement) du programme,
- de consultation des fichiers de description du problèmes, et
- de recherche, pour les deux premières solutions et au total.

On peut voir que, sur ce problème où elle est très contrainte, la localisation n'est pas loin de quadrupler la durée de l'exploration complète de l'espace des

[10] pour rappel : le modèle de base (contenant les indisponiblités), le placement imposé, la simultanéité, la consécutivité et l'extension aux locaux avec localisation imposée.

solutions. On verra que la différence est moins nette pour le problème réel, moins contraint. Par contre, l'utilisation d'une description réduite n'influence pas le temps de recherche d'un horaire sans locaux. Elle accélère seulement la consultation des données, de façon peu significative par rapport au temps total de résolution.

Notons que le tri préalable de la liste de cours par ordre de liberté croissante (*cf* section 5.3.2.1) est en fait indispensable. Une exécution du programme sans attribution de locaux, avec les cours dans un ordre défavorable (les moins contraints d'abord), n'avait encore trouvé aucune solution après 1020 secondes (17 minutes) de temps CPU. Par contre, après avoir trié les cours (ce qui n'a pris que 6,3 secondes), il lui a suffi de 16 secondes pour trouver et imprimer une solution au même problème.

5.3.5.2 Problème réel

Nous avons ensuite testé un problème réel : l'horaire du second semestre de l'institut d'informatique, limité au cycle de trois ans. Cette limitation a pour origine la possibilité pour les étudiants de seconde licence en deux ans de suivre n'importe quels cours de deuxième ou de troisième licence en trois ans, ce qui devrait imposer à ces cours d'être tous donnés à des moments différents. Or ils sont au nombre de 32, chacun d'une durée de deux heures. C'est donc impossible. En fait, cela oblige l'horaire officiel à planifier jusqu'à trois de ces cours en même temps, ce que notre modèle n'accepte pas.

Nous avons utilisé les contraintes réelles de l'année académique 88-89 qui entraient dans notre modèle : un grand nombre d'indisponibilités, quelques cours simultanés ou à plage fixée, aucune consécutivité, une seule localisation imposée. Quelques situations n'ont pu être prises en charge, principalement des activités par groupes devant avoir une plage attribuée mais pas de local : nous n'avons en effet pas la possibilité d'attribuer un local à certains cours et pas à d'autres.

Nous donnons en annexe la description du problème et la première solution qu'en trouve notre programme, avec localisation.

Notons que nous n'avons ici aucune idée du nombre total d'horaires possibles. Nous savons par contre que, à l'opposé du problème d'étude, l'introduction des locaux dans l'horaire multiplie les solutions.

Le tableau 3 reprend les temps de recherche, avec et sans localisation, pour les première, sixième et onzième et quinzième solutions.

La localisation de ce problème, peu contrainte, ne représente en temps qu'un surcoût de 25 %. On voit donc l'importance du niveau de liberté du problème sur son temps de résolution. On peut également constater que, une fois la première solution trouvée, les autres suivent au rythme de l'écriture dans le fichier des résultats : il suffit d'un changement minime pour passer d'une solution à l'autre.

Enfin, le tableau 4 donne une comparaison de l'espace mémoire utilisé pour construire les différents horaires.

Si l'on prend pour taille d'un problème d'horaire le nombre de cours à planifier, et si l'on considère l'occupation de mémoire du programme seul comme étant l'espace nécessaire à la résolution d'un problème de taille nulle, on peut voir dans ce tableau une évolution approximativement linéaire de la taille mémoire utilisée par rapport à la taille du problème. Si ceci est exact, les ressources qui seront les plus rapidement épuisées sont les piles locale et globale : un problème six fois plus grand que l'horaire de l'institut d'informatique serait déjà aux limites de la capacité de ces piles.

Par ailleurs, la localisation semble forte consommatrice de la pile locale et du "*trail*", ce qui s'explique naturellement par la multiplication des appels de procédure (pile locale) et des opérations de rétroparcours (*trail*) qu'elle implique.

5.3.6 Conclusions

L'approche *programmation logique* semble, par sa souplesse, fort prometteuse pour résoudre des problèmes d'horaires réels. Nous n'avons implémenté qu'un échantillon réduit de contraintes, mais cela nous a permis de voir avec quelle facilité on peut étendre le modèle de base.

En ce qui concerne les temps d'exécution, les résultats obtenus pour l'horaire de l'institut d'informatique, de taille plutôt modeste (46 cours) et peu contraint, ne nous permettent pas de préjuger des temps de résolution de gros problèmes : typiquement, plusieurs centaines de cours, et des horaires à peu près pleins pour les élèves, donc très contraints.

Chapitre 6

CONCLUSIONS GENERALES

Nous avons abordé dans ce mémoire la résolution du problème des horaires de cours par des méthodes d'utilisation récente.

Nous avons vu que la coloration de graphe classique est un modèle trop rigide pour la plupart des situations que l'on peut rencontrer. On peut toutefois espérer une certaine amélioration si l'on ajoute aux heuristiques de coloration des procédures spécifiques pour la prise en charge des contraintes qui ne sont pas modélisables dans le graphe.

Par contre, la coloration par fonction objectif (technique datant de quelques années seulement) présente une grande souplesse. La fonction objectif, qui n'est soumise à aucune condition de dérivabilité, peut en effet servir à représenter toutes sortes d'extensions au problème de base.

Enfin, la programmation logique, et en particulier le langage Prolog que nousavons utilisé, est un outil très prometteur pour la construction d'horaires, à condition de l'utiliser à bon escient. Il n'est en effet pas du tout approprié à l'implémentation de méthodes "numériques" (déterministes) comme le sont les heuristiques de coloration de graphe. Ainsi, les programmes Prolog que nous avons écrits pour la coloration CSG n'ont jamais pu colorer plus de 25 sommets. Par contre, nous avons obtenu de bons résultats avec une méthode non déterministe, parcourant l'espace des solutions pour trouver des horaires vérifiant les contraintes proposées.

L'analyse que nous avons faite dans ce mémoire peut, à notre avis, être approfondie dans deux directions.

D'une part, la caloration par fonction objectif peut faire l'objet d'un important travail de mise au point, de choix de paramètres, pour en assurer un fonctionnement optimal dans le cadre qui nous intéresse. Les multiples situations que l'on rencontre dans les problèmes réels devraient être analysés selon le point de vue "fonction objectif", les quelques idées que nous avons exposées ici devant être affinées et, éventuellement, étendues.

D'autre part, un logiciel de construction d'horaire, acceptant un grand nombre de contraintes et présentant une bonne ergonomie, pourrait être réalisé en programmation logique.

BIBLIOGRAPHIE

[1] Ivan BRATKO, Prolog Programming for Artificial Intelligence,
Addison-Wesley, 1986.

[2] Christopher John HOGGER, Introduction to Logic Programming,
Academic Press, 1984.

[3] Feliks KLUZNIAK, Stanislaw SZPAKOWICZ, Prolog for Programmers,
Academic Press, 1985.

[4] Pascal VAN HENTERIJK, Consistency Techniques in Logic Programming,
Thèse de doctorat en informatique présentée aux FNDP en 1987.

[5] Claude BERGE, Graphes et hypergraphes,
Dunod, 1973 ($2^{ème}$ édition).

[6] Nicos CHRISTOFIDES, Graph Theory - An Algorithmic Approach,
Academic Press, 1975.

[7] Ernesto BONOMI, Jean-Luc LUTTON, Le recuit simulé,
POUR LA SCIENCE n° 129 (juillet 1988).

[8] G. DREYFUS, Chaos et C.A.O. ou la méthode du "recuit simulé",
AFCET/INTERFACES n° 53 (mars 1987).

[9] F. GLOVER, Future paths for integer programming and links to artificial intelligence,
CAAI Report 85-8, University of Colorado, Boulder CO (1985).

[10] G. CAMPERS, Oswald HENKES, Jean-Paul LECLERCQ, Graph coloring heuristics : a survey, some new propositions and computational experiences on random and "Leighton's" graphs,
in "Operational Research '87", G.K.Rand (editor), Elsevier Science Publishers, © IFORS 1988 .

[11] M. CHAMS, A. HERTZ, D. de WERRA, Some experiments with simulated annealing for coloring graphs,
European Journal of Operational Research n° 32, 1987.

[12] Oswald HENKES, Graph coloring by an objective function applied to a school-timetable model,
in "Special Issue of Discrete Applied Mathematics on Timetabling and Chromatic Scheduling", à paraître en 1989.

[13] A. HERTZ, D. de WERRA, Using tabu search techniques for graph coloring,
Rapport interne - Ecole Polytechnique Fédérale de Lausanne (Suisse)

Notes

Année académique
1988-1989

Facultés Universitaires Notre-Dame de la Paix
Namur
Institut d'informatique

Le problème des horaires : analyse de moyens de résolution récents

---

Volume II : Annexes

Mémoire présenté pour l'obtention du grade de
Licencié en Sciences, option Informatique
par Emmanuel SIMONIS

Promoteur : Jean-Paul LECLERCQ

Annexe A

FONCTIONS ELEMENTAIRES

Fichier **elem.pro** (II-3)

```
%
%
%                          Fonctions elementaires a
%                        l'usage de programmes Prolog
%
```

Fichier elem.pro

```
%   [1]  Fonctions diverses
%        ~~~~~~~~~~~~~~~~~~

%   (1)  retract_all/1
%        ~~~~~~~~~~~~~
%        retract_all(Clause)
%   a pour fonction de
%        retirer de la base de faits toutes les clauses de la forme de
%        Clause (c'est-a-dire qui "matchent" avec Clause).

retract_all(Clause) :-
   retract(Clause),
   fail.

retract_all(_).

%   (2)  entiers_jsq/2
%        ~~~~~~~~~~~~~
%        entiers_jsq(Nombre,Liste)
%   est equivalent a
%        la liste "Liste" comprend les entiers de "Nombre" a 1, dans
%        cet ordre.
%   Note : pour un gain d'efficacite, entiers_jsq etant souvent appelee
%   avec le meme parametre "Nombre", elle effectue une "memorisation" de
%   ses resultats en creant en tete de procedure (instruction "asserta")
%   une clause
%               entiers_jsq(Nombre,[Nombre,...,2,1]) :- !.
%
%   Note : la procedure  entiers_jsq/2  utilise une fonction  aj_ent_jsq/3,
%        ou aj_ent_jsq(N,Liste,EntJsqN_et_Liste)
%   est equivalent a
%        la liste "EntJsqN_et_Liste" est la liste composee des entiers de
%        "N" a 1, dans cet ordre, suivis des elements de la liste "Liste".

entiers_jsq(N,Ent_Jsq_N) :-
   N > 0,
   aj_ent_jsq(N,[],Ent_Jsq_N),
   asserta( :-( entiers_jsq(N,Ent_Jsq_N) , ! ) ).

aj_ent_jsq(0,L,L) :- !.

aj_ent_jsq(N,Liste,EJN_et_Liste) :-
   Nm1 is N-1,
   aj_ent_jsq(Nm1,[N|Liste],EJN_et_Liste).
```

```
%   (3)  premier_par_cle/2
%        ******************
%        premier_par_cle(Liste_Cle_Objet,Cmax_O)
%   est equivalent a
%        si "Liste_Cle_Objet" est une liste non-vide de termes de la forme
%        [Cle,Objet], alors "Cmax_O" (de la forme [Cle_Max,Objet_CM] ) est
%        un element de la liste "Liste_Cle_Objet" tel que "Cle_Max" est
%        maximal parmi les "Cle" de la liste.


premier_par_cle([CO],CO).

premier_par_cle( [ [Cle,Objet] | Reste ], [Cle_Max,Objet_CM] ) :-
   premier_par_cle(Reste,[CM_Reste,O_CM_Reste]),
   ( Cle > CM_Reste, !,
        Cle_Max  = Cle,
        Objet_CM = Objet ;
      Cle_Max  = CM_Reste,
      Objet_CM = O_CM_Reste ).




%   [1]  Fonctions de listes
%        ~~~~~~~~~~~~~~~~~~~


%   (1)  member/2
%        ********
%        member(Objet,Liste)
%   est equivalent a
%        l'objet "Objet" est un element de la liste "Liste".
%
%   Note : member/2 peut etre utilise pour donner tous les elements d'une
%   liste, successivement, par backtracking.  Ceci interdit de definir la
%   premiere clause comme   "member(X,[X|_]) :- !."


member(X,[X|_]).

member(X,[_|Reste]) :-
   member(X,Reste).




%   (2)  delete/3
%        ********
%        delete(Objet,Liste,Liste_Moins_Objet)
%   est equivalent a
%        la liste "Liste_Moins_Objet" est la liste "Liste" ou une
%        occurence de "Objet" a ete enlevee.


delete(X,[X|Reste],Reste) :- !.

delete(X,[Y|Reste],[Y|Reste_Moins_X]) :-
   delete(X,Reste,Reste_Moins_X).
```

```
%   (3)  concat/3
%        ********
%        concat(Liste1,Liste2,Listes1_2)
%   est equivalent a
%        la liste "Listes1_2" est la concatenation des listes "Liste1"
%        et "Liste2" (elements de Liste1 suivis de ceux de Liste2).


concat([],Liste2,Liste2) :- !.

concat([Elem1|Reste1],Liste2,[Elem1|Reste1_Et_Liste2]) :-
   concat(Reste1,Liste2,Reste1_Et_Liste2).




%   (4)  intersection_lst/3
%        ******************
%        intersection_lst(Liste1,Liste2,Intersection)
%   est equivalent a
%        la liste "Intersection" contient les elements de la liste "Liste1"
%        qui sont aussi dans la liste "Liste2".
%   Note : "Intersection" sera sans doubles  SSI  "Liste1" l'est.


intersection_lst([],_,[]) :- !.

intersection_lst(_,[],[]) :- !.

intersection_lst([Elem1|Reste1],Liste2,Intersection) :-
   intersection_lst(Reste1,Liste2,Reste1_Inter_Liste2),
   ( member(Elem1,Liste2), !,                        % SI  Elem1 ds Liste2
       Intersection = [Elem1|Reste1_Inter_Liste2] ;    % ALORS  ...
       Intersection = Reste1_Inter_Liste2 ).           % SINON  ...




%   (5)  difference_lst/3
%        ****************
%        difference_lst(Liste1,Liste2,Difference)
%   est equivalent a
%        la liste "Difference" contient les elements de la liste "Liste1"
%        qui ne sont pas dans la liste "Liste2".
%   Note : "Difference" sera sana doubles  SSI  "Liste1" l'est.


difference_lst([],_,[]) :- !.

difference_lst([Elem1|Reste1],Liste2,Difference) :-
   difference_lst(Reste1,Liste2,Reste1_Moins_Liste2),
   ( member(Elem1,Liste2), !,                        % SI  Elem1 ds Liste2
       Difference = Reste1_Moins_Liste2 ;            % ALORS  ...
       Difference = [Elem1|Reste1_Moins_Liste2] ).   % SINON  ...
```

```
%   (6)  egalite_lst/2
%        ~~~~~~~~~~~~~~
%        egalite_lst(Liste1,Liste2)
%   est equivalent a
%        les liste "Liste1" et "Liste2" sont les memes a une permutation
%        pres, c'est-a-dire ont les memes elements.


egalite_lst([],[]) :- !.

egalite_lst([Elem1|Reste1],Liste2) :-
   delete(Elem1,Liste2,Reste2), ! ,    % echoue si Elem1 n'est pas dans Liste2,
                                       %   et Reste2 = (Liste2 \ Elem1)  sinon.
   egalite_lst(Reste1,Reste2).




%   (7)  lst_disjointes/2
%        ~~~~~~~~~~~~~~~~
%        lst_disjointes(Liste1,Liste2)
%   est equivalent a
%        les listes "Liste1" et "Liste2" n'ont pas d'element commun.


lst_disjointes([],_) :- !.

lst_disjointes(_,[]) :- !.

lst_disjointes([X|Reste],Liste) :-
   not( member(X,Liste) ),
   lst_disjointes(Reste,Liste).




%   (8)  liste_inversee/2
%        ~~~~~~~~~~~~~~~~
%        liste_inversee(Liste,Liste_Inverse)
%   est equivalent a
%        la liste "Liste_Inverse" est l'inverse de la liste "Liste", c'est-
%        a-dire que ses elements sont dans l'ordre inverse.
%
%   Note : liste_inversee/2 utilise une fonction inv_gener/3, ou
%        inv_gener(Reste,Fin_Inverse,Inverse)
%   est equivalent a
%        la liste "Inverse" est la liste obtenue en ajoutant la liste
%        "Fin_Inverse" au bout de l'inverse de la liste "Reste".


liste_inversee(Liste,Liste_Inv) :-
   inv_gener(Liste,[],Liste_Inv).


inv_gener([],I,I) :- !.

inv_gener([Elem|Reste],Fin_Inv,Inv) :-
   inv_gener(Reste,[Elem|Fin_Inv],Inv).
```

```
%   [2]  Fonctions d'ensembles
%        ^^^^^^^^^^^^^^^^^^^^^^

%   Representation des ensembles :
%   ''''''''''''''''''''''''''''''
%   Les ensembles sont representes par des listes triees (selon l'ordre total
%   standard) et sans doubles.
%   L'ordre standard (tel que decrit par le manuel utilisateur de C-Prolog,
%   version 1.5) est le siuvant :
%      - variables, en ordre standard (en gros, la plus ancienne d'abord ; le
%   nom des variables n'a aucune influence) ;
%      - references a la base de connaissances (en gros, en ordre d'age) ;
%      - les nombres, de "moins l'infini" a "plus l'infini" ;
%      - les atomes, en ordre lexicographique ASCII ;
%      - les termes complexes, d'abord par ordre d'arite (nombre d'arguments),
%   puis selon le nom du foncteur principal, puis selon les arguments de gauche
%   a droite.




%   (1)  element_ens/2
%        ^^^^^^^^^^^^^
%        element(Objet,Ensemble)
%   est equivalent a
%        l'objet Objet est un element de l'ensemble Ensemble.
%
%   Note : Cette fonction ne peut servir pour obtenir, par backtracking,
%   tous les elements d'un ensemble.  Pour ce faire, on utilisera la
%   fonction member/2.


element_ens(X,[X|Reste]) :- !.

element_ens(X,[Prem|Reste]) :-
   X @> Prem,
   element_ens(X,Reste).




%   (2)  intersection_ens/3
%        ^^^^^^^^^^^^^^^^^^
%        intersection_ens(Ensemble1,Ensemble2,Intersection)
%   est equivalent a
%        l'ensemble "Intersection" contient les elements de l'ensemble
%        "Ensemble1" qui sont aussi dans l'ensemble "Ensemble2".


intersection_ens([],_,[]) :- !.

intersection_ens(_,[],[]) :- !.

intersection_ens([Elem|Reste1],[Elem|Reste2],Inter) :-
   !,
   intersection_ens(Reste1,Reste2,Inter_Reste),
   Inter = [Elem|Inter_Reste].

intersection_ens([Elem1|Reste1],[Elem2|Reste2],Inter) :-
   Elem1 @< Elem2, !,
```

```
    intersection_ens(Reste1,[Elem2|Reste2],Inter) ;
    intersection_ens([Elem1|Reste1],Reste2,Inter) .




%   (3)  difference_ens/3
%        ****************
%        difference_ens(Ensemble1,Ensemble2,Difference)
%   est equivalent a
%        l'ensemble "Difference" contient les elements de l'ensemble
%        "Ensemble1" qui ne sont pas dans l'ensemble "Ensemble2".


difference_ens([],_,[]) :- !.

difference_ens(Reste,[],Reste) :- !.

difference_ens([Elem|Reste1],[Elem|Reste2],Difference) :-
   !,
   difference_ens(Reste1,Reste2,Difference).

difference_ens([Elem1|Reste1],[Elem2|Reste2],Difference) :-
   Elem1 @< Elem2, !,
      difference_ens(Reste1,[Elem2|Reste2],Reste1_Moins_Ens2),
      Difference = [Elem1|Reste1_Moins_Ens2] ;
    difference_ens([Elem1|Reste1],Reste2,Difference) .




%   (4)  egalite_ens/2
%        *************
%        egalite_ens(Ensemble1,Ensemble2)
%   est equivalent a
%        les ensembles "Ensemble1" et "Ensemble2" sont identiques (ont les
%        memes elements).


egalite_ens(Ens,Ens).




%   (5)  union_ens/3
%        ***********
%        union_ens(Ensemble1,Ensemble2,Union)
%   est equivalent a
%        l'ensemble "Union" est l'union des ensembles "Ensemble1" et
%        "Ensemble2".


union_ens([],E,E) :- !.

union_ens(E,[],E) :- !.

union_ens([Elem|Reste1],[Elem|Reste2],Union) :-
   !,
   union_ens(Reste1,Reste2,Union_Reste),
   Union = [Elem|Union_Reste].
```

```
union_ens([Elem1|Reste1],[Elem2|Reste2],Union) :-
   Elem1 @< Elem2, !,
       union_ens(Reste1,[Elem2|Reste2],Union_Reste),
       Union = [Elem1|Union_Reste] ;
     union_ens([Elem1|Reste1],Reste2,Union_Reste),
     Union = [Elem2|Union_Reste].
```

Annexe B

CONSTRUCTION D'HORAIRES
(programme logique)

Fichier mDO.pro

```
%
%                          Fonctions de
%                       manipulation des donnees
%                       (description d'un horaire)
%                         pour le modele de base
%
%
%


%   Ces fonctions renferment l'information sur la syntaxe du fichier de
%  donnees.  Celle-ci se base sur les operateurs definis ci-dessous.


%  Descripteurs des "objets" manipules
%  ''''''''''''''''''''''''''''''''''''

        :- op(950,xfx,[ est_la_plage,
                        est_le_cours,
                        est_le_prof,
                        est_la_classe,
                        est_le_local ]).


%  Descripteurs du probleme
%  ''''''''''''''''''''''''

        :- op(950,xfx,[ est_donne_par               % Cours (...) Prof
                      , est_suivi_par               % Cours (...) Classe
                      , est_indisponible_pour       % Prof  (...) Plage
                      , est_dans_le_groupe          % Cours (...) <nom>
                      , est_consecutive_a           % Plage (...) Plage
                      , doit_etre_consecutif_a      % Cours (...) Cours
                      , doit_etre_donne_en_plage    % Cours (...) Plage
                      , est_localise_dans           % Cours (...) local
                      , a_une_capacite_de           % Local (...) n
                      , a_un_effectif_de            % Classe (...) n
                      ]).


%   Ces operateurs s'utilisent dans des phrases dont la syntaxe est la
%  suivante :
%
%    - descripteurs des "objets" manipules :
%
%          <nom d'un objet>  <operateur>  <description de l'objet> .
%
%  exemples :
%      dupont    est_le_prof   'Jacques Dupont'.
%      c4B       est_la_classe '4eme annee B'.
%      math_4B_3 est_le_cours  'Mathematiques - 4e annee B (heure 3)'.
%      math_4B_4 est_le_cours  'Mathematiques - 4e annee B (heure 4)'.
%      chim_4B_1 est_le_cours  'Chimie - 4e annee B (heure 1)'.
%      mardi_3e  est_la_plage  'mardi 3e heure (10h40-11h30)'.
%      mardi_4e  est_la_plage  'mardi 4e heure (11h40-12h30)'.
%      labo_sc   est_le_local  'laboratoire de sciences'.
%
%
%
%    - descripteurs du probleme :
%
%   soit association d'objets :
```

```
%                    <nom d'un objet>  <operateur>  <nom d'un objet> .
%  soit qualification d'objets :
%                    <nom d'un objet>  <operateur>  <qualificatif> .
%
% exemples :
%     math_4B_3  est_donne_par            dupont.
%     math_4B_3  est_suivi_par            c4B.
%     dupont     est_indisponible_pour    mardi_4e.
%     mardi_4e   est_consecutive_a        mardi_3e.
%     math_4B_4  doit_etre_consecutif_a   math_4B_3.
%     chim_4B_1  doit_etre_donne_en_plage mardi_4e.
%     chim_4B_1  est_localise_dans        labo_sc.
%     loc25      a_une_capacite_de        28.
%     c4B        a_un_effectif_de         19.
%
% cas particulier :
%     math_4B_3  est_dans_le_groupe  gsl.
%   Cette clause signale que le cours math_4B_3 fait partie d'un groupe de
%   cours simultanes, appele "gsl" en l'occurence.  Il faut remarquer que
%   le groupe ne doit pas etre defini autrement (pas de clause du type
%   "est_le_groupe", par exemple).
%
%
%   Les noms des objets devraient etre UNIQUES (pour eviter tout risque de
% confusion).  Ce seront (de preference) des atomes prolog, c'est-a-dire des
% suites de caracteres d'un seul tenant (pas de blancs), pouvant contenir
% des lettres (majuscules ou minuscules), des chiffres et des "underscore"
% (caractere "_"), et devant OBLIGATOIREMENT commencer par une lettre
% minuscule.
%   Les descriptions des objets sont des suites de caracteres quelconques,
% encadrees de "quotes" (ou apostrophes : "'").
%   Les "qualificatifs" sont generalement des entiers.




%  [1]  Fonctions destinees au traitement du probleme
%       ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~



% <a>  Elements du probleme
%      ''''''''''''''''''''



%  (1)  lire_donnees_mDO/1
%       """"""""""""""""""
%       lire_donnees_mDO(Nom_Fichier)
%   a pour fonction de
%       (re)consulter le fichier "Nom_Fichier", c'est-a-dire integrer dans
%       la base de connaissances les clauses (descriptives de l'horaire a
%       traiter) qui s'y trouvent.


lire_donnees_mDO(Nom_Fichier) :-
   [-Nom_Fichier].
```

```
%   (2)  plage_horaire_mDO/1
%        ********************
%        plage_horaire_mDO(Plage)
%   est equivalent a
%        "Plage" est une plage horaire.
%
%   Note : Cette procedure donne, par backtracking, toutes les plages horaires.


plage_horaire_mDO(Plage) :-
   est_la_plage(Plage,_).




%   (3)  local_mDO/1
%        ************
%        local_mDO(Local)
%   est equivalent a
%        "Local" est un local.
%
%   Note : Cette procedure donne, par backtracking, tous les locaux.


local_mDO(Local) :-
   est_le_local(Local,_).




%   (4)  liste_cours_mDO/2
%        ******************
%        liste_cours_mDO(Liste_Cours,A_Trier)
%   est equivalent a
%        la liste "Liste_Cours" est la liste des cours a placer dans la
%        grille horaire.
%        Si "A_Trier" = 'tri', alors cette liste est triee dans un ordre
%        de liberte croissante (de sorte que les sommets susceptibles
%        d'etre plus difficiles a placer soient en tete).


liste_cours_mDO(LCours_Triee,tri) :-
   !,
   bagof( Plage, Descr^est_la_plage(Plage,Descr), Lst_Plages ),
   length(Lst_Plages,Nb_Plages),
   bagof( [Liberte,Cours],
          ( Descr^est_le_cours(Cours,Descr),
            estimation_liberte(Nb_Plages,Cours,Liberte) ),
          Liste_LC ),
   sort(Liste_LC,Liste_LC_Triee),
   bagof( Cours, Lib^member([Lib,Cours],Liste_LC_Triee), LCours_Triee).

liste_cours_mDO(LCours,_) :-
   bagof(Cours,Descr^est_le_cours(Cours,Descr),LCours).


estimation_liberte(Nb_Plages,Cours,Liberte) :-
```

```
                %%%>>>>>>>>>>>>>>>>>>>>>  Attribution de plage a priori
   ( doit_etre_donne_en_plage(Cours,_), !,    % SI plage imposee
        Nb_Plages_Autor is 1 ;                % ALORS une plage autorisee
                                              % SINON decompte des indisponib.
                %%%>>>>>>>>>>>>>>>>>>>>>  Contraintes d'indisponibilite
      ( est_donne_par(Cours,LProfs),
        ( setof( Plage,
                Prof^( member(Prof,LProfs),
                       est_indisponible_pour(Prof,Plage) ),
                Liste_Plages_Interd ), !,
          length(Liste_Plages_Interd,Nb_Pl_Interd) ;
          Nb_Pl_Interd is 0
        ),
        Nb_Plages_Autor is Nb_Plages - Nb_Pl_Interd
      )
   ),
                %%%>>>>>>>>>>>>>>>>>>>>>>>  Contraintes de simultaneite
   ( est_dans_le_groupe(Cours,Nom_Gr_Sim), !,
        bagof( CSim, est_dans_le_groupe(CSim,Nom_Gr_Sim), Gr_Sim),
        length(Gr_Sim,Taille_Gr_Sim) ;
      Taille_Gr_Sim = 0 ),
                %%%>>>>>>>>>>>>>>>>>>>>>>>  Contraintes de consecutivite
   ( bagof( C,
           ( doit_etre_consecutif_a(Cours,C) ;
             doit_etre_consecutif_a(C,Cours) ),
           Lst_Contr_Cons ),
     !, length(Lst_Contr_Cons,Nb_Contr_Cons) ;
     Nb_Contr_Cons is 0 ),
                %%%>>>>>>>>>>>>>>>>>>>>>>>  Totalisation
   Liberte is Nb_Plages_Autor - Taille_Gr_Sim - 2 * Nb_Contr_Cons.




% <b>  Modele de base
%      ,,,,,,,,,,,,,,,



%   (5)  cours_plage_incompat_mDO/2
%        """""""""""""""""""""""""""
%        cours_plage_incompat_mDO(Cours,Plage)
%   est equivalent a
%        le cours "Cours" et la plage horaire "Plage" sont incompatibles,
%        en ce sens que l'horaire ne peut attribuer cette plage a ce cours.


cours_plage_incompat_mDO(Cours,Plage) :-
   est_donne_par(Cours,L_Profs),
   member(Prof,L_Profs),         % echec de "est_indisp..." => essai autre prof
   est_indisponible_pour(Prof,Plage).




%   (6)  cours_independants_mDO/2
%        """""""""""""""""""""""""
%        cours_independants_mDO(Cours1,Cours2)
%   est equivalent a
%        les cours "Cours1" et "Cours2" sont independants, en ce sens que
```

```
%       l'horaire peut attribuer une plage horaire a l'un sans tenir compte
%       de celle attribuee a l'autre (il est possible de donner ces cours en
%       meme temps).
%
%   Notes : *) Si "Cours1" = "Cours2", la procedure echoue.
%           *) Cette procedure est prevue pour dire si deux cours donnes sont
%   independants, elle ne convient pas pour trouver un cours independant d'un
%   cours donne.


cours_independants_mDO(Cours,Cours) :-
   !, fail.

cours_independants_mDO(Cours1,Cours2) :-
   est_donne_par(Cours1,Profs_1),
   est_suivi_par(Cours1,Classes_1),
   est_donne_par(Cours2,Profs_2),
   est_suivi_par(Cours2,Classes_2), !,
   lst_disjointes(Classes_1,Classes_2), !,
   lst_disjointes( Profs_1,  Profs_2 ).




% <c>  Attribution de plage a priori
%      ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,


%   (7)  plage_imposee_mDO/2
%        ====================
%        plage_imposee_mDO(Cours,Plage)
%   est equivalent a
%        la plage "Plage" est imposee au cours "Cours".


plage_imposee_mDO(Cours,Plage) :-
   doit_etre_donne_en_plage(Cours,Plage).




% <d>  Contraintes de simultaneite
%      ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,


%   (8)  groupe_de_simultanes_mDO/2
%        ==========================
%        groupe_de_simultanes_mDO(Cours,Groupe)
%   est equivalent a
%        le cours "Cours" est dans le groupe de cours simultanes "Groupe".


groupe_de_simultanes_mDO(Cours,Groupe) :-
   est_dans_le_groupe(Cours,Groupe).
```

```
% (e) Contraintes de consecutivite
%     ''''''''''''''''''''''''''''''

% (9) plages_consec_mDO/2
%     """""""""""""""""""""
%     plages_consec_mDO(Premiere_Plage,Deuxieme_Plage)
% est equivalent a
%     la plage "Premiere_Plage" est consecutive a la plage "Deuxieme_Plage".

plages_consec_mDO(Plage_1,Plage_2) :-
   est_consecutive_a(Plage_2,Plage_1).

% (10) cours_consec_mDO/2
%      """""""""""""""""""""
%      cours_consec_mDO(Premier_Cours,Deuxieme_Cours)
% est equivalent a
%      le cours "Premier_Cours" doit etre consecutif au cours
%      "Deuxieme_Cours".

cours_consec_mDO(Cours_1,Cours_2) :-
   doit_etre_consecutif_a(Cours_2,Cours_1).

% (f) Attribution des locaux
%     ''''''''''''''''''''''''

% (11) cours_local_compat_mDO/2
%      """""""""""""""""""""""""""
%      cours_local_incompat_mDO(Cours,Local)
% est equivalent a
%      le cours "Cours" et le local "Local" sont compatibles, en ce sens
%      que la classe qui suit ce cours n'est pas trop nombreuse pour la
%      capacite de ce local.
%
% Note : Cette procedure peut donner, par backtracking, tous les locaux
% compatibles avec un cours donne.

cours_local_compat_mDO(Cours,Local) :-
   est_suivi_par(Cours,L_Classes),
   effectif_total(L_Classes,0,Effectif),
   a_une_capacite_de(Local,Capacite),      % donne tous les locaux, par backtr.
   Effectif =< Capacite.

effectif_total([],E,E).
effectif_total([Classe|Reste],Eff_Partiel,Effectif) :-
```

```
   a_un_effectif_de(Classe,Eff_Classe),
   Eff_Partiel_Plus is Eff_Partiel + Eff_Classe,
   effectif_total(Reste,Eff_Partiel_Plus,Effectif).



% <g>  Contraintes de localisation
%      ''''''''''''''''''''''''''



%  (12)  local_impose_mDO/2
%        """"""""""""""""""
%        local_impose_mDO(Cours,Local)
%  est equivalent a
%        le cours "Cours" doit etre donne dans le local "Local", suite
%        a une localisation a priori.


local_impose_mDO(Cours,Local) :-
   est_localise_dans(Cours,Local).



%  [2]  Fonctions destinees a l'affichage des resultats
%       ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~



%  (1)  caract_plage_mDO/2
%       """"""""""""""""""
%       caract_plage_mDO(Plage,Descr_Plage)
%  est equivalent a
%       la chaine de caracteres "Descr_Plage" est la description de la
%       plage horaire "Plage".


caract_plage_mDO(Plage,Descr_Plage) :-
   est_la_plage(Plage,Descr_Plage).



%  (2)  caract_cours_mDO/2
%       """"""""""""""""""
%       caract_cours_mDO(Cours,Descr_Cours,Liste_Profs,Liste_Classes)
%  est equivalent a
%       la chaine de caracteres "Descr_Cours" est la description du cours
%       "Cours", la liste "Liste_Profs" est la liste des professeurs qui
```

```
%       donnent ce cours, la liste "Liste_Classes" est la liste des classes
%       qui suivent ce cours.


caract_cours_mDO(Cours,Descr_Cours,Liste_Profs,Liste_Classes) :-
   est_le_cours(Cours,Descr_Cours),
   est_donne_par(Cours,Liste_Profs),
   est_suivi_par(Cours,Liste_Classes).




%   (3)  caract_classe_mDO/2
%        ********************
%        caract_classe_mDO(Classe,Descr_Classe)
%   est equivalent a
%        la chaine de caracteres "Descr_Classe" est la description de la
%        classe "Classe".


caract_classe_mDO(Classe,Descr_Classe) :-
   est_la_classe(Classe,Descr_Classe).




%   (4)  caract_prof_mDO/2
%        ******************
%        caract_prof_mDO(Prof,Descr_Prof)
%   est equivalent a
%        la chaine de caracteres "Descr_Prof" est la description du
%        professeur "Prof".


caract_prof_mDO(Prof,Descr_Prof) :-
   est_le_prof(Prof,Descr_Prof).




%   (5)  caract_local_mDO/2
%        *******************
%        caract_local_mDO(Prof,Descr_Local)
%   est equivalent a
%        la chaine de caracteres "Descr_Local" est la description du
%        local "Local".


caract_local_mDO(Local,Descr_Local) :-
   est_le_local(Local,Descr_Local).
```

Fichier mRE.pro

```
%
%                    Fonctions de manipulation
%                          d'un horaire
%                    (construction/consultation
%                          et impression)
%
%
```

```
%   Les fonctions de ce module utilisent des fonctions du fichier 'elem.pro'
%  (fonctions de base).
```

```
%   Fonctions de manipulation de l'horaire
%   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
%   (1)  init_grille_mRE/0
%        ^^^^^^^^^^^^^^^^^
%        init_grille_mRE
%   a pour fonction de
%        initialiser la representation de l'horaire.

init_grille_mRE :-
   retract_all(horaire(_,_)),
   retract_all(localisation(_,_)).
```

```
%   (2)  attribuer_plage_mRE/2
%        ^^^^^^^^^^^^^^^^^^^^^
%        attribuer_plage_mRE(Cours,Plage_Hor)
%   a pour fonction de
%           - au premier passage, inserer dans l'horaire l'association entre
%        le cours "Cours" et la plage "Plage_Hor" ;
%           - lors d'un backtracking, defaire cette association et echouer.

attribuer_plage_mRE(Cours,Plage_Hor) :-
   asserta(horaire(Cours,Plage_Hor)).

attribuer_plage_mRE(Cours,Plage_Hor) :-
   retract(horaire(Cours,Plage_Hor)),
   fail.
```

```
%   (3)  attribuer_local_mRE/2
%        ^^^^^^^^^^^^^^^^^^^^^
%        attribuer_local_mRE(Cours,Local)
%   a pour fonction de
%           - au premier passage, inserer dans l'horaire l'association entre
%        le cours "Cours" et le local "Local" ;
%           - lors d'un backtracking, defaire cette association et echouer.
```

```
attribuer_local_mRE(Cours,Local) :-
   asserta(localisation(Cours,Local)).

attribuer_local_mRE(Cours,Local) :-
   retract(localisation(Cours,Local)),
   fail.




%   (4)  plage_attribuee_mRE/2
%        ^^^^^^^^^^^^^^^^^^^^^
%        plage_attribuee_mRE(Cours,Plage_Hor)
%   est equivalent a
%        le cours "Cours" est prevu pour la plage "Plage_Hor".


plage_attribuee_mRE(Cours,Plage_Hor) :-
   horaire(Cours,Plage_Hor).




%   (5)  local_occupe_mRE/2
%        ^^^^^^^^^^^^^^^^^^
%        local_occupe_mRE(Local,Plage)
%   est equivalent a
%        le local "Local" est occupe pendant la plage horaire "Plage".


local_occupe_mRE(Local,Plage) :-
   horaire(Crs,Plage),
   localisation(Crs,Local).




%   (6)  local_attribue_mRE/2
%        ^^^^^^^^^^^^^^^^^^^^
%        local_attribue_mRE(Cours,Local)
%   est equivalent a
%        le cours "Cours" est prevu dans le local "Local".


local_attribue_mRE(Cours,Local) :-
   localisation(Cours,Local).




%   Fonctions relatives a l'ecriture de l'horaire dans le fichier des resultats
%   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
%   (1)  ecrire_solution/6
%        ******************
%        ecrire_solution_PL(Temps_Recherche,Demande_Locaux)
%   a pour fonction de
%        ecrire dans le fichier de sortie ("output stream") courant la
%        derniere solution trouvee, avec
%           "Temps_Recherche" : temps ecoule depuis le debut de la recherche ;
%           "Demande_Locaux" : indicateur de demande de l'attribution des
%        locaux (atome "loc" si elle a ete demandee, autre sinon) ;


ecrire_solution_PL(T,Locaux) :-
   write('   Solution trouvee apres '),
   write(T),
   write(' secondes CPU'), nl,
   write('-----------------------------------------------'), nl,
   ecrire_horaire(Locaux),
   write('==============================================='), nl.



ecrire_horaire(Locaux) :-
   caract_plage_@DO(P,Descr_P),                    % backtr. sur les plages
   write('Cours prevus pour  '),
   write(Descr_P), nl,
   ecrire_hor_plage(P,Locaux),
   fail.

ecrire_horaire(_).



ecrire_hor_plage(P,loc) :-
   plage_attribuee_@RE(C,P),
   local_attribue_@RE(C,L),
   caract_cours_@DO(C,Cours,Profs,Classes),
   write('   '), write(Cours), nl,
   write('      local : '),
     caract_local_@DO(L,Local),
     write(Local), nl,
   write('      prof(s) : '),
     ecrire_liste_profs(Profs), nl,
   write('      classe(s) : '),
     ecrire_liste_classes(Classes), nl,
   fail.

ecrire_hor_plage(_,loc) :- !.

ecrire_hor_plage(P,_) :-
   plage_attribuee_@RE(C,P),
   caract_cours_@DO(C,Cours,Profs,Classes),
   write('   '), write(Cours), nl,
   write('      prof(s) : '),
     ecrire_liste_profs(Profs), nl,
   write('      classe(s) : '),
     ecrire_liste_classes(Classes), nl,
   fail.

ecrire_hor_plage(_,_).
```

```
ecrire_liste_profs([]) :- write('-'), !.

ecrire_liste_profs([P]) :-
   !,
   ( caract_prof_mDO(P,Prof), ! ; Prof is P ),
   write(Prof).

ecrire_liste_profs([P|Reste]) :-
   caract_prof_mDO(P,Prof),
   write(Prof),
   write(', '),
   ecrire_liste_profs(Reste).



ecrire_liste_classes([C]) :-
   !,
   ( caract_classe_mDO(C,Classe), ! ; Classe is C ),
   write(Classe).

ecrire_liste_classes([C|Reste]) :-
   caract_classe_mDO(C,Classe),
   write(Classe),
   write(', '),
   ecrire_liste_classes(Reste).
```

Fichier grillePL.pro

```
%
%                        Implementation en Prolog
%                     d'une methode non-deterministe
%                  (exploration exhaustive par retroparcours)
%                        de construction d'horaires
%


%   Les fonctions definies ci-dessous utilisent des fonctions definies dans
%  les fichiers suivants :
%               'elem.pro'    (fonctions de base)
%               'mDO.pro'     (manipulation des donnees, suffixe '_mDO')
%               'mRE.pro'     (manipulation des resultats, suffixe '_mRE',
%                              et ecriture des solutions, sans suffixe)




%   (1)  independance/2
%        ***************
%        independance(Cours,Liste_Cours)
%   est equivalent a
%        le cours "Cours" est independant des cours de la liste
%        "Liste_Cours", deux cours etant independants ssi les professeurs
%        qui les donnent sont differents et les classes qui les suivent
%        sont differents.


independance(_,[]) :- !.

independance(Cours_a_placer,[Cours|Reste]) :-
   cours_independants_mDO(Cours_a_placer,Cours), !,
   independance(Cours_a_placer,Reste).




%   (2)  association_possible/2
%        ***********************
%        association_possible(Cours,Plage)
%   est equivalent a
%        le cours "Cours" peut etre donne a la plage "Plage" au vu des
%        choix d'horaire deja effectues, c'est-a-dire que ce cours est
%        independant de tous les cours qui sont deja prevus pour cette
%        plage.


association_possible(Cours,Plage) :-
   not( cours_plage_incompat_mDO(Cours,Plage) ),
   ( bagof(C,plage_attribuee_mRE(C,Plage),Cours_en_P), !,
        independance(Cours,Cours_en_P) ;
      true ).                          % Cas "rien de prevu en plage Plage"




%   (3)  choisir_plage/2
%        ****************
%        choisir_plage(Cours,Plage)
```

```
%   est equivalent a
%        la plage "Plage" est
%           - soit la seule plage autorisee au cours "Cours", au vu des
%        contraintes et des choix d'horaire deja effectues ;
%           - soit une plage quelconque si ce cours n'est pas contraint.
%        Si les contraintes ne peuvent etre toutes verifiees, la procedure
%        echoue.
%        Si le cours n'est pas contraint, la procedure donnera successivement
%        toutes les plages de l'horaire, sur backtracking.
%
%   Note : Les contraintes prises en compte ici sont les suivantes.
%      - La simultaneite de cours : si le cours "Cours" est simultane avec un
%   autre cours ayant deja une plage attribuee, on doit prendre cette plage.
%      - La consecutivite de cours : si le cours "Cours" doit SUIVRE le cours
%   "Cours_Prec", alors "Plage" doit etre la plage suivant celle associee a
%   "Cours_Prec" ; si Cours doit PRECEDER le cours "Cours_Suiv", alors "Plage"
%   doit etre la plage precedant celle associee a "Cours_Suiv".
%      - L'attribution a priori d'une plage a un cours.


choisir_plage(Cours,Plage) :-
   setof(
      Pl_Imposee,
      (
        plage_imposee_mDO(Cours,Pl_Imposee)
        ;
        Cours_Prec^
        Plage_Prec^(
                    cours_consec_mDO(Cours_Prec,Cours),
                    plage_attribuee_mRE(Cours_Prec,Plage_Prec),
                    ( plages_consec_mDO(Plage_Prec,Pl_Imposee), ! ;
                                        Pl_Imposee = impossibilite  )
                   ) ;
        Cours_Suiv^
        Plage_Suiv^(
                    cours_consec_mDO(Cours,Cours_Suiv),
                    plage_attribuee_mRE(Cours_Suiv,Plage_Suiv),
                    ( plages_consec_mDO(Pl_Imposee,Plage_Suiv), ! ;
                                        Pl_Imposee = impossibilite  )
                   ) ;
        Groupe^
        Cours_Sim^(
                    groupe_de_simultanes_mDO(Cours,Groupe),
                    groupe_de_simultanes_mDO(Cours_Sim,Groupe),
                    Cours \== Cours_Sim,
                    plage_attribuee_mRE(Cours_Sim,Pl_Imposee)
                   )
      ),
      L_Pl_Imposees ),
   !,
   L_Pl_Imposees = [Plage],         % Si plusieurs plages imposees, echec.
   not( Plage = impossibilite ).    % Si impossibilite, echec.


choisir_plage(_,Plage) :-
   plage_horaire_mDO(Plage).




%   (4)  planifier_temps/2
```

```
%       ********************
%       planifier_temps(Cours,Plage)
%   a pour fonction de
%       - Au premier passage, trouver une plage qui puisse etre associee
%       au cours "Cours", en fonction des choix d'horaire deja effectues
%       et des contraintes du probleme.  Si c'est possible, inserer cette
%       association cours-plage dans l'horaire ; "Plage" est la plage
%       associee au cours.
%       - Lors d'un backtracking, defaire l'association de plage a ce cours
%       et en etablir une autre, s'il en existe (sinon, echoue).


planifier_temps(Cours,PH) :-
   choisir_plage(Cours,PH),        % generateur pour le backtr., si pas contr.
   association_possible(Cours,PH), % echec => essai d'une autre plage
   attribuer_plage_mRE(Cours,PH).  % backtr : defait l'association puis echoue




%   (5)  planifier_lieu/2
%        ****************
%       planifier_lieu(Cours,Plage)
%   a pour fonction de
%       - au premier passage, associer un local au cours "Cours", en fonction
%       de la plage "Plage" qui lui a ete attribuee et des contraintes du
%       probleme, si c'est possible (sinon, echoue) ;
%       - lors d'un backtracking, defaire l'association d'un local a ce cours
%       et en etablir une autre, s'il en existe (sinon, echoue).
%
%   Note : si le cours a ete localise a priori, le local ainsi attribue est
%   retenu.  Dans ce cas, s'il y a backtracking, la procedure echoue.


planifier_lieu(Cours,Plage) :-
   local_impose_mDO(Cours,Local), !,
   not( local_occupe_mRE(Local,Plage) ),
   attribuer_local_mRE(Cours,Local).      % backtr : defait la maj puis echoue

planifier_lieu(Cours,Plage) :-
   local_mDO(Local),
   not( local_occupe_mRE(Local,Plage) ),
   cours_local_compat_mDO(Cours,Local),
   attribuer_local_mRE(Cours,Local).      % backtr : defait la maj puis echoue




%   (6)  planifier/2
%        ***********
%       planifier(Liste_Cours,Demande_Locaux)
%   a pour fonction de
%       placer dans l'horaire les cours de la liste "Liste_Cours", si c'est
%       possible (echoue sinon).
%       Si "Demande_Locaux", indicateur de demande d'attribution de
%       locaux, est l'atome "avec_loc", la procedure attribue a chaque
%       cours une plage horaire et un local.
%       Si "Demande_Locaux" est l'atome "sans_loc", la procedure gere
%       seulement les plages horaires.
%       Dans les autres cas, la procedure echoue.
```

```
planifier([],_).

planifier([Cours|Reste],avec_loc) :-
   planifier_temps(Cours,Plage_Attr),
   planifier_lieu(Cours,Plage_Attr),
   planifier(Reste,avec_loc).

planifier([Cours|Reste],sans_loc) :-
   planifier_temps(Cours,_),
   planifier(Reste,sans_loc).




%   (7)  ecrire_entete/6
%        ################
%        ecrire_entete(Temps_Initialisation,Nom_Fich_Donnees,
%                      Demande_Locaux,Demande_Tri,Temps_Minimal_Backtr)
%   a pour fonction de
%        ecrire dans le fichier de sortie ("output stream") courant l'entete
%        du document donnant les resultats, avec
%           "Temps_Initialisation" : temps pris pour initialiser la recherche
%        (consultation des donnees, saisie de la liste des cours,...) ;
%           "Nom_Fich_Donnees" : nom du fichier de donnees ;
%           "Demande_Locaux" : indicateur de demande de l'attribution des
%        locaux (atome "loc" si elle a ete demandee, autre sinon) ;
%           "Demande_Tri" : indicateur de demande de tri prealable des cours
%        selon leur liberte croissante (atome "tri" si le tri a ete demande,
%        autre sinon) ;
%           "Temps_Minimal_Backtr" : temps de recherche au-dela duquel le
%        backtracking n'est plus provoque.


ecrire_entete(T_Init,Fich_Donn,Locaux,Tri,T_Backtr) :-
   write('^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^'), nl,
   write('  Resolution du probleme decrit dans le fichier  '), nl,
   write('                     '), write(Fich_Donn),          nl,
   write('          ('),
   ( Locaux = loc, !, write('avec') ; write('sans') ),
                          write(' assignation de locaux)'), nl,
   write('-----------------------------------------------'), nl,
   write('  Tri preliminaire des cours : '),
   ( Tri = tri, !, write('EFFECTUE') ; write('non demande') ), nl,
   write('  Temps de preparation : '),
                           write(T_Init),write(' s  CPU.'), nl,
   write('  Temps minimal de recherche : '),
                          write(T_Backtr),write(' s  CPU.'), nl,
   nl,
   write('^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^'), nl.




%   (8)  message_fin/2
%        ##############
%        message_fin(Fichier,Condition)
%   a pour fonction de
%        ecrire a l'ecran et dans le fichier de nom "Fichier" le message
```

```
%       de fin de travail, dependant de la condition de fin :
%          - si "Condition" est l'atome "arret_sur_temps", la recherche a
%       ete arretee sur base du temps ecoule, lequel depassait le temps
%       minimum de backtracking ;
%          - si "Condition" est l'atome "arret_sur_echec", le programme a
%       examine toutes les possibilites et sorti toutes les solutions
%       trouvees sur le fichier de resultats.
%       La procedure ferme le fichier "Fichier" apres y avoir ecrit le
%       message.


message_fin(Fich_Res,arret_sur_temps) :-
   retract(debut_rech(Tps_Deb)),
   Tps_Tot is cputime - Tps_Deb,
   retract(nb_sol(NS)),
   write('Le temps minimal de recherche est atteint.'), nl,
   write('Nombre de solutions trouvees : '), write(NS), nl,
   close(Fich_Res),
   tell(user),
   write('Le temps minimal de recherche que vous avez demande est atteint.'),
   nl, write('   (temps de recherche : '), write(Tps_Tot), write(' s CPU.)'),
   nl, write(NS), write(' solutions ont ete trouvees.'),
   nl.

message_fin(Fich_Res,arret_sur_echec) :-
   retract(debut_rech(Tps_Deb)),
   Tps_Tot is cputime - Tps_Deb,
   retract(nb_sol(NS)),
   write('Toutes les possibilites ont ete explorees.'), nl,
   write('Temps total de recherche     : '), write(Tps_Tot),
   write(' s CPU'), nl,
   write('Nombre de solutions trouvees : '), write(NS), nl,
   close(Fich_Res),
   tell(user),
   write('Toutes les possibilites ont ete explorees.'), nl,
   write('Temps total de recherche     : '), write(Tps_Tot),
   write(' s CPU'), nl,
   ( NS = 0 , !, write('Aucune solution n''a ete trouvee.')    ;
     NS = 1 , !, write('Une seule solution a ete trouvee.')    ;
           write(NS), write(' solutions ont ete trouvees.')     ), nl.




%   (9)  grille_PL/5
%        ~~~~~~~~~~~~
%        grille_PL(Nom_Fich_Donnees,Nom_Fich_Resultats,
%                  Demande_Locaux,Demande_Tri,Temps_Minimal_Backtr)
%   a pour fonction de
%        resoudre le probleme d'horaires decrit dans le fichier dont
%        le nom est "Nom_Fich_Donnees", et ecrire les solutions trouvees
%        dans le fichier dont le nom est "Nom_Fich_Resultats".
%        Modalites de recherche :
%           - attribution des locaux (en plus des plages horaires)
%        si "Demande_Locaux" est l'atome "loc" ;
%           - tri preliminaire des cours, en ordre de liberte croissante,
%        de sorte que les cours semblant les plus difficiles a placer le
%        soient d'abord, si "Demande_Tri" est l'atome "tri" ;
%           - "Temps_Minimal_Backtr" est le temps CPU minimal de backtracking,
%        au dela duquel on ne provoque plus le backtracking pour chercher une
%        solution supplementaire.
```

```
grille_PL(Fich_Donn,Fich_Res,Locaux,Tri,Tps_CPU_Pour_Backtr) :-
   Deb_Init is cputime,
   asserta(nb_sol(0)),
   lire_donnees_mDO(Fich_Donn),
   liste_cours_mDO(Liste_Cours,Tri),
   init_grille_mRE,
   tell(Fich_Res),
   T_Init is cputime-Deb_Init,
   ecrire_entete(T_Init,Fich_Donn,Locaux,Tri,Tps_CPU_Pour_Backtr),

   D is cputime,   assert(debut_rech(D)),

   (Locaux=loc, !,
      planifier(Liste_Cours,avec_loc) ;
      planifier(Liste_Cours,sans_loc) ),

   Temps_CPU_Ecoule is cputime - D,

   ecrire_solution_PL(Temps_CPU_Ecoule,Locaux),
   increm_compteur_sol,
   fail_sur_condition(Temps_CPU_Ecoule,Tps_CPU_Pour_Backtr),
   message_fin(Fich_Res,arret_sur_temps).


grille_PL(_,Fich_Res,_,_,_) :-
   message_fin(Fich_Res,arret_sur_echec).




%   fail_sur_condition(A,B)   echoue si A < B et reussit sinon.
%   ***********************


fail_sur_condition(Tps_Ecoule,Tps_Backtr) :-
   Tps_Ecoule < Tps_Backtr, !,
      fail ;
   true.




%   increm_compteur_sol   augmente d'une unite l'argument de la premiere
%   ********************  clause "nb_sol/1" de la base de connaissances,
%                         et echoue en cas de backtracking.


increm_compteur_sol :-
   retract(nb_sol(N)),
   Np1 is N + 1,
   asserta(nb_sol(Np1)), !.
```

Fichier horairePL.pro

```
%
%                          Chargement
%                      du programme Prolog
%                   de construction d'horaires
%                   (optique non-deterministe)
%
%


:- nl, nl, write('          ... Veuillez patienter ...'), nl, nl.


:-  [-'elem.pro'].         % Fonctions elementaires
:-  [-'mDO.pro'].          % Module des donnees
:-  [-'mRE.pro'].          % Module des resultats
:-  [-'grillePL.pro'].     % Fonctions principales


message_init :-
   nl, nl,
   write('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~'),nl,
   write('   Pour construire un horaire (par recherche non-deterministe),'),nl,
   write('utilisez la fonction  grille_PL/5  selon le format suivant :   '),nl,
   nl,
   write('  ?-  grille_PL( <input>, <output>, <loc>, <tri>, <temps> ).   '),nl,
   nl,
   write('avec   <input> : nom du fichier de donnees                     '),nl,
   write('      <output> : nom du fichier des resultats                  '),nl,
   write('         <loc> : demande d''attribution des locaux ; ecrivez le'),nl,
   write('  mot "loc" pour construire un horaire avec locaux, ecrivez une'),nl,
   write('  lettre quelconque sinon (plages seulement)                   '),nl,
   write('         <tri> : demande d''optimisation ; ecrivez le mot "tri"'),nl,
   write('  pour composer l''horaire en commencant par les cours les plus'),nl,
   write('  difficiles a placer, ecrivez une lettre quelconque sinon     '),nl,
   write('       <temps> : provision de temps pour le backtracking (apres'),nl,
   write('  avoir trouve une solution, la fonction en cherchera une autre'),nl,
   write('  si le temps CPU ecoule est inferieur a <temps>, en secondes).'),nl,
   nl,
   write('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~'),nl,
   nl.


:-  message_init.
```

Annexe C

COLORATION DE GRAPHES

```
%
%
%                          Type abstrait GRAPHE
%
%
%
```

Fichier taGR.pro

```
%  Notes :
%    Ce type est defini par 10 fonctions, dont le nom est suffixe par les
%  caracteres "_taGR".  La representations du graphe utilise des clauses
%  dont le nom est lui aussi suffixe par "_taGR".
%    La definition du type utilise les fonctions de base definies dans le
%  fichier "elem.pro".


%  Les fonctions du type GRAPHE sont les suivantes.  Elles ne peuvent
%  manipuler qu'un seul graphe.
%
%  Constructeurs :
%
%      init_graphe_taGR
%          > initialise le graphe
%
%      ajout_sommet_taGR( Sommet )
%          > ajoute un sommet au graphe
%          > echoue si le sommet existe
%
%      signaler_groupe_taGR( Groupe, Liste_Cours )
%          > declare un groupe de cours simultanes, representes par
%          >    un seul sommet
%          > echoue si le groupe est deja declare
%
%      def_adjacents_taGR( Sommet, Liste_Adjacents )
%          > definit les adjacents d'un sommet
%          > echoue si le sommet n'existe pas
%          >    ou  si les adjacents sont deja definis
%
%      attribuer_couleur_taGR( Sommet, Couleur )
%          > attribue une couleur a un sommet
%          > echoue si le sommet n'existe pas
%          >    ou  si ses adjacents ne sont pas definis
%          >    ou  si il est deja colore
%
%  Selecteurs :
%
%      ens_sommets_taGR( Ensemble_Sommets )
%          > donne l'ensemble des sommets du graphe
%
%      adjacents_taGR( Sommet, Ensemble_Adjacents )
%          > donne l'ensemble des adjacents d'un sommet
%          > echoue si les adjacents du sommet ne sont pas definis
%
%      couleur_attribuee_taGR( Sommet, Couleur )
%          > donne la couleur attribuee a un sommet
%          > echoue si le sommet n'existe pas
%          >    ou  si ses adjacents ne sont pas definis
%          >    ou  si il n'est pas colore
%
%      couleurs_interdites_taGR( Sommet, Ensemble_Couleurs )
%          > donne l'ensemble des couleurs interdites a un sommet
%          > echoue si le sommet n'existe pas
```

```
%          )    ou  si ses adjacents ne sont pas definis
%          )    ou  si le sommet est colore
%
%      coloration_taGR( Coloration )
%          ) donne la coloration du graphe, sous forme d'une liste de termes
%          )   de la forme "couleur(Sommet,Couleur)"
%          ) echoue s'il existe des sommets non colores
%
%      groupe_taGR( Groupe, Liste_Cours )
%          ) donne la liste des cours simultanes correspondant a un
%          )   sommet-groupe donne
%          ) echoue si ce n'est pas un groupe
%
%      coherence_taGR
%          ) reussit SSI le graphe est "coherent", c'est-a-dire ssi
%          )     - les adjacents d'un sommet sont definis
%          )                              et  sont des sommets existants,
%          )     - les adjacents d'un sommet ont ce sommet pour adjacent,
%          )     - aucun sommet n'est adjacent a lui-meme
%



init_graphe_taGR :-
   abolish(sommet_taGR,3),
   abolish(groupe_simult_taGR,2).



ajout_sommet_taGR( Sommet ) :-
   not sommet_taGR(Sommet,_,_),                % echoue si Sommet deja defini
   assert(sommet_taGR(Sommet,inc,inc)).



signaler_groupe_taGR( Groupe, Liste_Cours ) :-
   not groupe_simult_taGR(Groupe,_),           % echoue si Groupe deja declare
   assert(groupe_simult_taGR(Groupe,Liste_Cours)).



def_adjacents_taGR( Sommet, L_Adj ) :-
   sort(L_Adj,LT_Adj),
   retract(sommet_taGR(Sommet,inc,inc)),       % echoue si a deja des adjacents
   assert(sommet_taGR(Sommet,LT_Adj,[[]])).                %-  ou n'existe pas



attribuer_couleur_taGR( Sommet, Couleur ) :-
   sommet_taGR(Sommet,EnsAdj,[_]),     % echoue si non defini ou si deja colore
   ajouter_interdictions_taGR(EnsAdj,Couleur),
   retract(sommet_taGR(Sommet,EnsAdj,[_])),
   assert(sommet_taGR(Sommet,EnsAdj,Couleur)).

ajouter_interdictions_taGR([],_) :- !.
ajouter_interdictions_taGR([S|Reste],Couleur) :-
   sommet_taGR(S,EA,_),                        % le sommet S existe
```

```
   not(EA = inc),                              % et est bien defini
   ajouter_interdictions_taGR(Reste,Couleur),
   ( sommet_taGR(S,_,[_]), !,                          % SI S n'est pas colore
      retract(sommet_taGR(S,Adj_S,[Ens_Coul_Interd])),   % ALORS mise-a-jour
      union_ens([Couleur],Ens_Coul_Interd,Nouv_ECI),
      assert(sommet_taGR(S,Adj_S,[Nouv_ECI]))   ;
     true ).                                              % SINON ne rien faire



ens_sommets_taGR( Ens_Sommets ) :-
   setof(Sommet,X^(Y^sommet_taGR(Sommet,X,Y)),Ens_Sommets).



adjacents_taGR( Sommet, Ens_Adj ) :-
   sommet_taGR(Sommet,Ens_Adj,_),
   not( Ens_Adj = inc ).



couleur_attribuee_taGR( Sommet, Couleur ) :-
   sommet_taGR(Sommet,_,Couleur),
   not(Couleur = inc),             % echoue si le sommet n'est pas bien defini
   not(Couleur = [_]).             % echoue si Couleur est une liste, cad
                                   %       si Sommet n'est pas colore



couleurs_interdites_taGR( Sommet, Ens_Coul_Interd ) :-
   sommet_taGR(Sommet,_,[Ens_Coul_Interd]).



coloration_taGR( Coloration ) :-
   not(sommet_taGR(_,_,[_])),       % echoue si il existe un sommet non colore
   not(sommet_taGR(_,inc,inc)),     % echoue si il existe un sommet mal defini
   bagof(couleur(Somm,Coul), X^sommet_taGR(Somm,X,Coul), Coloration).



groupe_taGR( Groupe, Liste_Cours ) :-
   groupe_simult_taGR(Groupe,Liste_Cours).



coherence_taGR :-
   sommet_taGR(Somm,Ens_Adj,_),
   member(Somm_Adj,Ens_Adj),
   assert(arc_taGR(Somm,Somm_Adj)),
   fail.
coherence_taGR :-
   coherence_par_arete_taGR,
   not(sommet_taGR(_,inc,_)).
```

```
coherence_par_arete_taGR :-
   arc_taGR(S_Orig,S_Dest), !,
        %%  SI il reste un arc non examine
        %%  ALORS
   (arc_taGR(S_Dest,S_Orig), !,
        %%    SI son "inverse" existe (arete coherente)
        %%    ALORS retirer les deux arcs, verifier la coherence du reste
   retract(arc_taGR(S_Orig,S_Dest)),
   retract(arc_taGR(S_Dest,S_Orig)),
   coherence_par_arete_taGR ;
        %%    SINON echec, apres avoir "nettoye" ce qui reste
   abolish(arc_taGR,2),
   fail );
        %%  SINON succes (toutes les aretes sont verifiees et coherentes)
   true.
```

Fichier graphe.pro

```
%
%                              Fonctions de
%                           manipulation de graphe
%                         pour des programmes Prolog
%
%
%


%   Note :
%   Ces fonctions utilisent :
%       - le type abstrait GRAPHE defini dans le fichier "taGR.pro" (les
%   fonctions definissant ce type sont suffixees par les caracteres "_taGR")
%       - les fonctions de base definies dans le fichier "elem.pro".




%   (1)  degre/2
%        ********
%


degre(Sommet,Degre) :-
   adjacents_taGR(Sommet,Adj),
   length(Adj,Degre).




%   (2)  sommet_degre_max/1
%        ******************
%


sommet_degre_max(Sommet) :-
   ens_sommets_taGR(Ens_Sommets),
   rech_sdm(Ens_Sommets,Sommet,_).


rech_sdm([Somm],Somm,Deg) :-
   !, degre(Somm,Deg).

rech_sdm([Somm|Reste],Somm_Deg_Max,Deg_Max) :-
   rech_sdm(Reste,SDM_Reste,DM_Reste),
   degre(Somm,Deg),
   ( Deg > DM_Reste, !,
        Deg_Max = Deg,
        Somm_Deg_Max = Somm ;
      Deg_Max = DM_Reste,
      Somm_Deg_Max = SDM_Reste ).




%   (3)  couleurs_autorisees/3
%        **********************
%        couleurs_autorisees(Nb_Couleurs,Sommet,Couleurs_Autorisees)
```

```
%   est equivalent a
%        si le sommet "Sommet" n'est pas encore colore,
%        alors, sur la palette des couleurs de 1 a "Nb_Couleurs", ce sommet
%        peut prendre les couleurs de l'ensemble "Couleurs_Autorisees", d'apres
%        les couleurs attribuees a ses adjacents
%        (si le sommet est deja colore, la fonction echoue).


couleurs_autorisees(NbCoul,Sommet,Ens_Coul_Autor) :-
   entiers_jsq(NbCoul,Ens_Coul),
   couleurs_interdites_taGR(Sommet,Ens_Coul_Interd),
   difference_ens(Ens_Coul,Ens_Coul_Interd,Ens_Coul_Autor).




%   (4)  sommet_liberte_min/1
%        """""""""""""""""""""
%        sommet_liberte_min(Sommet)
%   a pour fonction de
%        trouver un sommet "Sommet" de liberte minimale parmi les sommets
%        non-colores, c'est-a-dire dont le nombre de couleurs interdites est
%        maximal.
%        (Echoue s'il n'existe pas de sommet non-colore).


sommet_liberte_min(Sommet) :-
   ens_sommets_taGR(Ens_Sommets),
   rech_scm(Ens_Sommets,Sommet,Contrainte),
   Contrainte >= 0.          % Contrainte=-1 SSI tous les sommets sont colores


rech_scm([Somm],Somm,Contr) :-
   ( couleurs_interdites_taGR(Somm,EnsCoul), !,   % echoue ssi Somm est colore
        length(EnsCoul,Contr)                ;
      Contr is -1                            ).   % deja colore => a ignorer

rech_scm([Somm|Reste],Somm_Contr_Max,Contr_Max) :-
   rech_scm(Reste,SCM_Reste,CM_Reste),
   ( couleurs_interdites_taGR(Somm,EnsCoul), !,   % echoue ssi Somm est colore
        length(EnsCoul,Contr)                ;
      Contr is -1                            ),   % deja colore => a ignorer
   ( Contr > CM_Reste, !,
        Contr_Max = Contr,
        Somm_Contr_Max = Somm ;
      Contr_Max = CM_Reste,
      Somm_Contr_Max = SCM_Reste ).




%   (5)  adjacents_ordre2/2
%        """""""""""""""""""
%        adjacents_ordre2(Sommet,Liste_Adjacents_ordre2)
%   a pour fonction de
%        donner la liste "Liste_Adjacents_ordre2" des adjacents
%        d'ordre deux du sommet "Sommet".


adjacents_ordre2(Sommet,Liste_Adj2) :-
```

```
   adjacents_taGR(Sommet,Ens_Adj),
   bagof( Adj2,
          Adj^ Ens_Adj_de_Adj^( member(Adj,Ens_Adj),
                                adjacents_taGR(Adj,Ens_Adj_de_Adj),
                                member(Adj2,Ens_Adj_de_Adj) ),
          Liste_Adj2 ),
   !.                          % bagof echoue SSI Sommet n'a pas d'adj_2

adjacents_ordre2(_,[]).        % si bagof echoue, retourner la liste vide




%   (6)  sommet_degre2_max/1
%        ********************
%


sommet_degre2_max(Sommet) :-
   ens_sommets_taGR(Ens_Sommets),
   rech_sd2m(Ens_Sommets,Sommet,_).


rech_sd2m([Somm],Somm,Deg2) :-
   !, degre2(Somm,Deg2).

rech_sd2m([Somm|Reste],Somm_Deg2_Max,Deg2_Max) :-
   rech_sd2m(Reste,SD2M_Reste,D2M_Reste),
   degre2(Somm,Deg2),
   ( Deg2 > D2M_Reste, !,
        Deg2_Max = Deg2,
        Somm_Deg2_Max = Somm ;
      Deg2_Max = D2M_Reste,
      Somm_Deg2_Max = SD2M_Reste ).


degre2(Sommet,Degre2) :-
   adjacents_taGR(Sommet,L_Adj),
   somme_degres(L_Adj,0,Degre2).

somme_degres([],D2,D2) :- !.
somme_degres([Sommet|Reste],Somme_Part,Somme) :-
   degre(Sommet,Deg_Sommet),
   Somme_Part_Plus is Somme_Part + Deg_Sommet,
   somme_degres(Reste,Somme_Part_Plus,Somme).
```

Fichier clique.pro

```
%
%                          Fonctions de
%                       recherche d'une clique
%                      (algorithme heuristique)
%
%
```

```
%   Les fonctions decrites ci-dessous utilisent les fonctions des fichiers
% suivants :
%            'elem.pro'    (fonctions de base)
%            'taGR.pro'    (type abstrait GRAPHE - suffixe '_taGR')
%            'graphe.pro'  (manipulation du graphe).
%
%   Le type abstrait GRAPHE travaille sur un seul graphe.  Les fonctions de
% ce fichier ont pour but la recherche d'une clique (sous-graphe complet)
% de ce graphe, que l'on espere de taille maximale, selon un algorithme
% heuristique.
%
%   Un premier groupe de fonctions renferme entierement l'information sur la
% representation du sous-graphe utilise par notre heuristique.
```

```
%  [1]  Fonctions manipulant le sous-graphe
%       ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
%  Representation du sous-graphe
%  ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
%  Le sous-graphe est represente par des clauses de la forme
%              sommet_sg(Sommet,Ens_Adj),
%  ou "Sommet"  est un sommet du sous-graphe,
%     "Ens_Adj" est l'ensemble des adjacents de ce sommet dans le sous-graphe.
```

```
%  (1)  creer_sous_graphe/1
%       ********************
%       creer_sous_graphe(Ens_Sommets)
%  a pour fonction de
%       generer le sous-graphe engendre par les sommets de l'ensemble
%       "Ens_Sommets"   (ensemble : liste triee en ordre croissant et
%       sans doubles).
%       Le graphe de depart est le graphe courant du type abstrait GRAPHE.
%
%  Note : creer_sous_graphe/1 utilise les fonctions creer_clauses_sg/1 et
%  completer_clauses_sg/1, ou
%     creer_clauses_sg(Ens_Sommets)
%  a pour fonction de
%     creer une clause "sommet_sg/2" par sommet de l'ensemble "Ens_Sommets",
%     dont le second argument est une liste vide,
%  et
%     completer_clauses_sg(Ens_Sommets)
%  a pour fonction de
%     determiner les adjacents de chaque sommet dans le sous-graphe, et de
%     remplacer, dans la clause "sommet_sg/2" correspondante, la liste vide
%     par l'ensemble des adjacents.
```

```
creer_sous_graphe(Ens_Sommets) :-
   abolish(sommet_sg,2),
   creer_clauses_sg(Ens_Sommets),
   completer_clauses_sg(Ens_Sommets).


creer_clauses_sg([]).
creer_clauses_sg([Sommet|Reste]) :-
   assertz(sommet_sg(Sommet,inc)),
   creer_clauses_sg(Reste).

completer_clauses_sg(Ens_Sommets) :-
   retract(sommet_sg(S,inc)),
   adjacents_taGR(S,Ens_Adj_dans_GR),
   intersection_ens(Ens_Sommets,Ens_Adj_dans_GR,Ens_Adj_dans_sg),
   assertz(sommet_sg(S,Ens_Adj_dans_sg)),
   fail.
completer_clauses_sg(_).




%   (2)  reduire_sous_graphe/1
%        ''''''''''''''''''''''''
%        reduire_sous_graphe(Ens_Somm_Generateurs)
%   a pour fonction de
%        reduire le sous-graphe a son sous-graphe engendre par les sommets
%        de l'ensemble "Ens_Somm_Generateurs".


reduire_sous_graphe(Ens_Somm_Gener) :-
   retract(sommet_sg(S,Ens_Adj_ds_sg)),
   reduire_sous_graphe(Ens_Somm_Gener),
   member(S,Ens_Somm_Gener), !,                % SI S est un des generateurs
     intersection_ens(                         %   ALORS S est repris dans le
              Ens_Adj_ds_sg,Ens_Somm_Gener,    %        nouveau sous-graphe,
              Ens_Adj_ds_nouv_sg         ),    %        avec un ensemble
     assert(sommet_sg(S,Ens_Adj_ds_nouv_sg)).  %        d'adjacents corrige
                                               %   SINON on ne fait rien

reduire_sous_graphe(_).




%   (3)  sg_est_vide/0
%        ''''''''''''''
%        sg_est_vide
%   est equivalent a
%        le sous-graphe est vide.


sg_est_vide :-
   not sommet_sg(_,_).
```

```
%   (4)  adjacents_dans_sg/2
%        ^^^^^^^^^^^^^^^^^^^^
%        adjacents_dans_sg(Sommet,Ens_Adj)
%   est equivalent a
%        les adjacents, dans le sous-graphe, du sommet "Sommet" sont les
%        sommets de l'ensemble "Ens_Adj".


adjacents_dans_sg(Sommet,Ens_Adj) :-
   sommet_sg(Sommet,Ens_Adj).




%   (5)  sommet_deg_max_ds_sg/1
%        ^^^^^^^^^^^^^^^^^^^^^^^
%        sommet_deg_max_ds_sg(Sommet)
%   est equivalent a
%        le sommet "Sommet" est un sommet de degre maximal dans le
%        sous-graphe.


sommet_deg_max_ds_sg(Sommet) :-
   bagof( [D,S], Adj^(sommet_sg(S,Adj),length(Adj,D)), Liste_DS ),
   premier_par_cle(Liste_DS,[_,Sommet]).



%   [2]  Recherche d'une clique
%        ^^^^^^^^^^^^^^^^^^^^^^


%   (1)  recherche_clique/1
%        ^^^^^^^^^^^^^^^^^^^
%        recherche_clique(Clique)
%   est equivalent a
%        les sommets de la liste "Clique" sont les sommets d'une clique (ou
%        sous-graphe complet) du graphe traite par le type abstrait GRAPHE.
%
%   Note : cette fonction utilise la fonction constr_clique/2 definie ensuite.


recherche_clique(Clique) :-
   sommet_degre2_max(S),
   adjacents_taGR(S,Ens_Adj),
   creer_sous_graphe(Ens_Adj),
   constr_clique([S],Clique).




%   (2)  constr_clique/2
%        ^^^^^^^^^^^^^^^^
%        constr_clique(Clique_a_agrandir,Clique)
%   est equivalent a
%        SI les sommets de la liste "Clique_a_agrandir" forment une clique C1
%        du graphe, et
%        SI chaque sommet du sous-graphe courant est un adjacent de tous
%        les sommets de la clique C1,
```

```
%       ALORS les sommets de la liste "Clique" forment une clique C2 au moins
%       aussi grande que C1.
%
%   Note : on peut verifier que les deux conditions ci-dessus (SI ...) sont
%   etablies  - par la fonction  recherche_clique/1  et
%             - par la fonction  constr_clique/2 (si elles sont verifiees lors
%   de son appel),
%   avant chaque appel de constr_clique/2.
%   En outre,  constr_clique/2  reduit le sous-graphe d'au moins un sommet
%   avant de se rappeler recursivement.


constr_clique(Clique,Clique) :-
   sg_est_vide.

constr_clique(Clique_a_agrandir,Clique) :-
   sommet_deg_max_ds_sg(Somm),
   adjacents_dans_sg(Somm,Ens_Adj),
   reduire_sous_graphe(Ens_Adj),
   constr_clique([Somm|Clique_a_agrandir],Clique).
```

Fichier csg.pro

```
%
%                  Implementation en Prolog de la methode
%                        Complete Sub Graph (CSG)
%                                ----
%                              Version 2
%                     (basee sur un type abstrait GRAPHE)
%


%    Les fonction definies ci-dessous utilisent des fonctions definies dans
%  les fichiers
%              'elem.pro'   (fonctions de base)
%              'taGR.pro'   (type abstrait GRAPHE - suffixe '_taGR')
%              'graphe.pro' (fonctions de manipulation du graphe)
%              'clique.pro' (fonctions de recherche d'une clique)
%
%  Note : Le type abstrait GRAPHE ne manipule qu'un graphe.  Les fonctions
%  qui suivent s'appliquent a cet unique graphe, qui doit etre defini
%  prealablement a l'utilisation de  csg/1  en utilisant les fonctions
%  du type abstrait.




%  (1)  effectif_coul_parmi_adj2/3
%       ^^^^^^^^^^^^^^^^^^^^^^^^^^
%       effectif_coul_parmi_adj2(Sommet,Couleur,Effectif)
%  est equivalent a
%       l'entier "Effectif" est l'effectif de la couleur "Couleur" parmi
%       les adjacents d'ordre 2 du sommet "Sommet" (voir aussi description
%       de la fonction  couleur_presence_max_adj2/3 : ce comptage se fait
%       d'une maniere un peu particuliere).


effectif_coul_parmi_adj2(Sommet,Couleur,Effectif) :-
   adjacents_ordre2(Sommet,Liste_Adj2),
   bagof( Adj2,
          ( member(Adj2,Liste_Adj2),
            couleur_attribuee_taGR(Adj2,Couleur) ),
          Liste_Adj2_de_Coul ), !,
   length(Liste_Adj2_de_Coul,Effectif).

effectif_coul_parmi_adj2(_,_,0).




%  (2)  couleur_presence_max_adj2/3
%       ^^^^^^^^^^^^^^^^^^^^^^^^^^^
%       couleur_presence_max_adj2(Sommet,Coul_Autor,Couleur_Pres_Max)
%  est equivalent a
%       la couleur "Couleur_Pres_Max" est, parmi les couleurs de la liste
%       "Coul_Autor", celle qui est la plus representee parmi les adjacents
%       d'ordre 2 du sommet "Sommet".
%       Remarque : les adjacents d'ordre 2 d'un sommet S sont les sommets
%       accessibles a partir de S en suivant en chemin de longueur 2 (deux
%       aretes).  Ils sont comptes (avec leur couleur) autant de fois qu'il
```

```
%       n'y a de ces chemins de longueur 2.
%
%   Exemple : Dans le graphe ci-dessous,
%
%                   x1              - la couleur de x5 serait comptee 3 fois
%                                   comme "couleur d'adjacent d'ordre 2" du
%                /  |  \            sommet x1, une fois pour chacun des trois
%              a/   |b  \c          chemins de longueur 2 (a,d), (b,e), (c,f) ;
%              /    |    \          - les couleurs de x2, x3 et x4 seraient
%                                   comptees 2 fois comme "couleur d'adjacent
%            x2     x3    x4       d'ordre 2" du sommet x2, du fait des
%                                   chemins
%              \    |    /                 (a,a) et (d,d) pour x2,
%              d\   |e  /f                 (a,b) et (d,e) pour x3 et
%                \  |  /                   (a,c) et (d,f) pour x4.
%
%                   x5


couleur_presence_max_adj2(Sommet,Coul_Autor,Couleur_Pres_Max) :-
   bagof( [Effectif,Couleur],
          ( member(Couleur,Coul_Autor),
            effectif_coul_parmi_adj2(Sommet,Couleur,Effectif) ),
          Liste_EC ),
   premier_par_cle(Liste_EC,[_,Couleur_Pres_Max]).




%   (3)  colorer_sommet/4
%        ================
%        colorer_sommet(Sommet,Couleurs_Autor,Nb_Coul,Nouv_Nb_Coul)
%   a pour fonction de
%        donner une couleur au sommet "Sommet" du graphe, sachant
%        que : - les couleurs autorisees au sommet sont donnees par
%        la liste "Couleurs_Autor",
%              - le graphe est deja partiellement colore en "Nb_Coul"
%        couleurs ;
%        "Nouv_Nb_Coul" est le nombre de couleurs dans le graphe, apres
%        coloration du sommet.
%   Note : la methode de coloration utilisee est la 'Complete Sub Graph'.


colorer_sommet(Sommet,[Couleur],Nb_Coul,Nb_Coul) :-
   attribuer_couleur_taGR(Sommet,Couleur), !.

colorer_sommet(Sommet,[],Nb_Coul,Nouv_Nb_Coul) :-
   Nouv_Nb_Coul  is  Nb_Coul + 1,
   attribuer_couleur_taGR(Sommet,Nouv_Nb_Coul), !.

colorer_sommet(Sommet,Coul_Autor,Nb_Coul,Nb_Coul) :-
   couleur_presence_max_adj2(Sommet,Coul_Autor,Couleur_Pres_Max),
   attribuer_couleur_taGR(Sommet,Couleur_Pres_Max), !.




%   (4)  colorer_reste/3
%        ===============
%        colorer_reste(Sommets_A_Colorer,Nb_Coul,Nb_Coul_Final)
```

```
%   a pour fonction de
%        donner une couleur aux sommets de la liste "Sommets_A_Colorer"
%        du graphe, sachant qu'il existe deja une coloration partielle
%        (en "Nb_Coul" couleurs).  "Nb_Coul_Final" est le nombre de
%        couleurs de la coloration au complet.
%    Note : la methode de coloration utilisee est la 'Complete Sub Graph'.


colorer_reste([],NC,NC) :- !.

colorer_reste(Sommets_A_Colorer,Nb_Coul,Nb_Coul_Final) :-
   sommet_liberte_min(Sommet_LM),
   couleurs_autorisees(Nb_Coul,Sommet_LM,Coul_Autor),
   colorer_sommet(Sommet_LM,Coul_Autor,Nb_Coul,Nouv_Nb_Coul),
   difference_ens(Sommets_A_Colorer,[Sommet_LM],Reste_A_Colorer),
   colorer_reste(Reste_A_Colorer,Nouv_Nb_Coul,Nb_Coul_Final), !.




%   (5)  colorer_clique/2
%        ''''''''''''''''''
%        colorer_clique(Clique,Nb_Couleurs_Clique)
%   a pour fonction de
%        donner une couleur differente a chacun des sommets de la liste
%        "Clique", et de donner comme valeur a "Nb_Couleurs_Clique" le
%        nombre de couleurs utilisees.


colorer_clique([],0) :- !.

colorer_clique([Sommet|Reste],Nb_Couleurs) :-
   colorer_clique(Reste,Nb_Coul_Reste),
   Nb_Couleurs  is  Nb_Coul_Reste + 1,
   attribuer_couleur_taGR(Sommet,Nb_Couleurs), !.




%   (6)  colorer_graphe/2
%        ''''''''''''''''''
%        colorer_graphe(Clique,Nb_Couleurs)
%   a pour fonction de
%        colorer le graphe, en en connaissant une clique "Clique" ("Clique"
%        est un ensemble - liste triee sans doubles - de sommets) ;
%        "Nb_Couleurs" est le nombre de couleurs de cette coloration.
%    Note : la methode de coloration utilisee est la 'Complete Sub Graph'.


colorer_graphe(Clique,Nb_Couleurs) :-
   colorer_clique(Clique,Nb_Coul_Clique),
   ens_sommets_taGR(Ens_Sommets),
   difference_ens(Ens_Sommets,Clique,Sommets_A_Colorer),
   colorer_reste(Sommets_A_Colorer,Nb_Coul_Clique,Nb_Couleurs), !.
```

```
%   (7)  csg/1
%        ******
%        csg(Nb_Couleurs)
%   a pour fonction de
%        trouver une coloration du graphe ; "Nb_Couleurs" est la taille de
%        cette coloration.
%
%   Note : on peut obtenir la coloraton du graphe en utilisant la fonction
%   coloration_taGR/1 du type abstrait GRAPHE.


csg(Nb_Couleurs) :-
   recherche_clique(Clique),
   sort(Clique,Clique_Triee),
   colorer_graphe(Clique_Triee,Nb_Couleurs).
```

Fichier lf.pro

```
%
%                    Implementation en Prolog de la methode
%                              Largest First (LF)
%                           de coloration de graphes
%                                    ----
%                                  Version 2
%                        (basee sur un type abstrait GRAPHE)
%



%    Les fonction definies ci-dessous utilisent des fonctions definies dans
%   les fichiers
%               'elem.pro'    (fonctions de base)
%               'taGR.pro'    (type abstrait GRAPHE - suffixe '_taGR')
%               'graphe.pro'  (fonctions de manipulation du graphe)
%
%   Note : Le type abstrait GRAPHE ne manipule qu'un graphe.  Les fonctions
%   qui suivent s'appliquent a cet unique graphe, qui doit etre defini
%   prealablement a l'utilisation de  lf/2  en utilisant les fonctions
%   du type abstrait.



%   (1) choix_couleur/4
%       ^^^^^^^^^^^^^^^
%       choix_couleur(Coul_Autorisees,Nb_Coul_Existantes,Couleur,Nouv_Nb_Coul)
%   est equivalent a
%       si la liste "Coul_Autorisees" n'est pas vide, la couleur "Couleur"
%       en est le premier element et "Nouv_Nb_Coul" est egal a
%       "Nb_Coul_Existantes" ;
%       sinon, la couleur "Couleur" est une nouvelle couleur
%       ("Nb_Coul_Existantes" + 1) et "Nouv_Nb_Coul" est "Nb_Coul_Existantes"
%       augmente de 1.


choix_couleur([],N,N_p_1,N_p_1) :-
   N_p_1  is  N + 1.

choix_couleur([Couleur|_],N,Couleur,N).



%   (2) colorer_LF/3
%       ^^^^^^^^^^^^
%       colorer_LF(LDS_Triee,Nb_Coul_Exist,Nb_Final_Coul)
%   a pour fonction de
%       colorer, selon la methode LF, les sommets (donnes avec leurs degres,
%       sous la forme [Degre,Sommet]) de la liste "LDS_Triee" triee par ordre
%       de degre decroissant,
%       sachant que d'autre sommets ont deja ete colores avec une des
%       "Nb_Coul_Exist" couleurs deja existantes ;
%       "Nb_Final_Coul" est le nombre final de couleurs utilisees.


colorer_LF([],N,N).

colorer_LF([ [_,Sommet] | Reste ],Nb_Coul_Exist,Nb_Final_Coul) :-
```

```
   couleurs_autorisees(Nb_Coul_Exist,Sommet,Coul_Autor),
   choix_couleur(Coul_Autor,Nb_Coul_Exist,Couleur,Nouv_Nb_Coul),
   attribuer_couleur_taGR(Sommet,Couleur),
   colorer_LF(Reste,Nouv_Nb_Coul,Nb_Final_Coul).




%   (3) lf2/2
%       ~~~~~
%       lf2(Coloration,Nb_Couleurs)
%   est equivalent a
%       la liste "Coloration", dont les elements sont de la forme
%       "couleur(Sommet,Couleur)", donne la coloration du graphe represente
%       dans le module taGR obtenu selon la methode LF ;
%       "Nb_Couleurs" est le nombre de couleurs utilisees par cette
%       coloration.


lf2(Coloration,Nb_Couleurs) :-
   ens_sommets_taGR(Ens_Sommets),
   bagof( [DegNeg,Sommet],
          Degre^( member(Sommet,Ens_Sommets),
                  degre(Sommet,Degre),
                  DegNeg is -Degre          ),
          Liste_DS ),
   sort(Liste_DS,LDS_Triee),
   colorer_LF(LDS_Triee,1,Nb_Couleurs),
   coloration_taGR(Coloration).
```

Annexe D

HORAIRES DE TEST

Les problèmes d'horaire qui nous ont servi pour tester notre programme non-déterministe (*cf* annexe B) sont listés ci-après, avec un fichier de résultats pour chacun.

On trouvera :

- la description complète, c'est-à-dire avec tout ce qui se rapporte aux locaux, de l'horaire 0 : fichier `horOL` (`L` pour locaux) ;
- les résultats obtenus en exécutant le programme sur le problème `horOL`, en demandant une localisation (`L`) et un tri préliminaire des cours (`T`) : fichier `horOL_LT` ;
- la description de l'horaire de l'institut d'informatique : fichier `horInfo` ;
- un horaire avec localisation et tri préliminaire des cours, répondant au problème `horInfo` : fichier `horI_LT1` (une seule solution).

Dans l'entête des fichiers de résultats, on notera les lignes

```
(avec assignation de locaux)
```

et

```
Tri preliminaire des cours : EFFECTUE,
```

qui signalent (respectivement) qu'une localisation et un tri des cours ont été demandés. L'entête mentionne également :

- le temps de préparation, comprenant essentiellement la consultation des données et la formation de la liste des cours (et son tri éventuel) ;
- le temps minimal de recherche, au-delà duquel le programme s'arrête dès qu'il a trouvé une solution.

En outre, chaque solution est précédée du temps CPU écoulé entre le début de la recherche et le moment où cette solution a été trouvée. Il comprend le temps d'écriture, dans le fichier des résultats, des solutions trouvées auparavant.

En fin de fichier, on trouve la cause de l'arrêt (soit tout a été essayé, soit le temps minimal de recherche est atteint), le temps de recherche total et le nombre de solutions trouvées.

Fichier horOL

```
%
%         Horaire 0   (probleme d'etude)
%
%         Version 'horOL' (avec locaux)
%               * modele de base + placement a priori
%                               + simultaneite
%                               + consecutivite
%               * attribution des locaux (capacites, localisation imposee)
%               * 2 solutions
%               * cours donnes dans un ordre defavorable (le meme que hor0)


%---  Plages

pl1 est_la_plage 'plage 1'.
pl2 est_la_plage 'plage 2'.
pl3 est_la_plage 'plage 3'.
pl4 est_la_plage 'plage 4'.
pl5 est_la_plage 'plage 5'.

pl2 est_consecutive_a pl1.
pl3 est_consecutive_a pl2.
pl5 est_consecutive_a pl4.



%---  Locaux

loc1  est_le_local  '*( 1 )*'.
loc2  est_le_local  '*( 2 )*'.
loc3  est_le_local  '*( 3 )*'.

loc1  a_une_capacite_de  40.
loc2  a_une_capacite_de  25.
loc3  a_une_capacite_de  10.



%---  Classes

clA .est_la_classe  '--- A ---'.
clB  est_la_classe  '--- B ---'.
clC  est_la_classe  '--- C ---'.

clA  a_un_effectif_de  20.
clB  a_un_effectif_de  15.
clC  a_un_effectif_de  10.



%---  Professeurs

prof1  est_le_prof  '== 1 =='.
prof2  est_le_prof  '== 2 =='.
prof3  est_le_prof  '== 3 =='.
prof4  est_le_prof  '== 4 =='.



%---  Cours

crC2_p1   est_le_cours  'Cours C - 2'.
```

```
crA2_p4   est_le_cours  'Cours A - 2'.
crB1_p2   est_le_cours  'Cours B - 1'.
crAB_p2   est_le_cours  'Cours A+B'.
crBC_p3   est_le_cours  'Cours B+C'.
crC1_p3   est_le_cours  'Cours C - 1'.
crAC_p3   est_le_cours  'Cours A+C'.
crA3_p24  est_le_cours  'Cours A - 3'.
crB2_p12  est_le_cours  'Cours B - 2'.
crA1_p1   est_le_cours  'Cours A - 1'.
conf      est_le_cours  'Conferences'.


%---  Cours - Profs - Classes

crA1_p1   est_suivi_par  [clA].
crA1_p1   est_donne_par  [prof1].

crA2_p4   est_suivi_par  [clA].
crA2_p4   est_donne_par  [prof4].

crA3_p24  est_suivi_par  [clA].
crA3_p24  est_donne_par  [prof2,prof4].

crAB_p2   est_suivi_par  [clA,clB].
crAB_p2   est_donne_par  [prof2].

crAC_p3   est_suivi_par  [clA,clC].
crAC_p3   est_donne_par  [prof3].

crB1_p2   est_suivi_par  [clB].
crB1_p2   est_donne_par  [prof2].

crB2_p12  est_suivi_par  [clB].
crB2_p12  est_donne_par  [prof1,prof2].

crBC_p3   est_suivi_par  [clB,clC].
crBC_p3   est_donne_par  [prof3].

crC1_p3   est_suivi_par  [clC].
crC1_p3   est_donne_par  [prof3].

crC2_p1   est_suivi_par  [clC].
crC2_p1   est_donne_par  [prof1].

conf      est_suivi_par  [].
conf      est_donne_par  [].


%---  Profs - Plages   (3 indisponibilites)

prof1  est_indisponible_pour  pl2.

prof3  est_indisponible_pour  pl3.

prof4  est_indisponible_pour  pl4.


%---  Plage imposee   (1 contrainte)
```

```
conf  doit_etre_donne_en_plage  pl5.


%---  Cours simultanes   (2 contraintes)

crA1_p1   est_dans_le_groupe  sim_1.
crB1_p2   est_dans_le_groupe  sim_1.

crA3_p24  est_dans_le_groupe  sim_5.
crC2_p1   est_dans_le_groupe  sim_5.


%---  Cours consecutifs   (1 contrainte)

crA2_p4  doit_etre_consecutif_a  crA1_p1.


%---  Locaux imposes   (2 contraintes)

crB1_p2  est_localise_dans  loc1.
conf     est_localise_dans  loc1.
```

Fichier horOL_LT

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  Resolution du probleme decrit dans le fichier
                    horOL
          (avec assignation de locaux)
------------------------------------------------------------
  Tri preliminaire des cours : EFFECTUE
  Temps de preparation : 8.13334 s  CPU.
  Temps minimal de recherche : 900 s  CPU.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   Solution trouvee apres 182.667 secondes CPU
------------------------------------------------------------
Cours prevus pour  plage 1
   Cours C - 1
      local : *( 3 )*
      prof(s) : == 3 ==
      classe(s) : --- C ---
   Cours B - 1
      local : *( 1 )*
      prof(s) : == 2 ==
      classe(s) : --- B ---
   Cours A - 1
      local : *( 2 )*
      prof(s) : == 1 ==
      classe(s) : --- A ---
Cours prevus pour  plage 2
   Cours B+C
      local : *( 2 )*
      prof(s) : == 3 ==
      classe(s) : --- B ---, --- C ---
   Cours A - 2
      local : *( 1 )*
      prof(s) : == 4 ==
      classe(s) : --- A ---
Cours prevus pour  plage 3
   Cours A+B
      local : *( 1 )*
      prof(s) : == 2 ==
      classe(s) : --- A ---, --- B ---
Cours prevus pour  plage 4
   Cours B - 2
      local : *( 2 )*
      prof(s) : == 1 ==, == 2 ==
      classe(s) : --- B ---
   Cours A+C
      local : *( 1 )*
      prof(s) : == 3 ==
      classe(s) : --- A ---, --- C ---
Cours prevus pour  plage 5
   Cours C - 2
      local : *( 3 )*
      prof(s) : == 1 ==
      classe(s) : --- C ---
   Cours A - 3
      local : *( 2 )*
      prof(s) : == 2 ==, == 4 ==
      classe(s) : --- A ---
   Conferences
      local : *( 1 )*
      prof(s) : -
      classe(s) :
============================================================
```

```
   Solution trouvee apres 334.933 secondes CPU
-------------------------------------------------
Cours prevus pour  plage 1
   Cours C - 1
      local : #( 3 )#
      prof(s) : == 3 ==
      classe(s) : --- C ---
   Cours B - 1
      local : #( 1 )#
      prof(s) : == 2 ==
      classe(s) : --- B ---
   Cours A - 1
      local : #( 2 )#
      prof(s) : == 1 ==
      classe(s) : --- A ---
Cours prevus pour  plage 2
   Cours B+C
      local : #( 1 )#
      prof(s) : == 3 ==
      classe(s) : --- B ---, --- C ---
   Cours A - 2
      local : #( 2 )#
      prof(s) : == 4 ==
      classe(s) : --- A ---
Cours prevus pour  plage 3
   Cours A+B
      local : #( 1 )#
      prof(s) : == 2 ==
      classe(s) : --- A ---, --- B ---
Cours prevus pour  plage 4
   Cours B - 2
      local : #( 2 )#
      prof(s) : == 1 ==, == 2 ==
      classe(s) : --- B ---
   Cours A+C
      local : #( 1 )#
      prof(s) : == 3 ==
      classe(s) : --- A ---, --- C ---
Cours prevus pour  plage 5
   Cours C - 2
      local : #( 3 )#
      prof(s) : == 1 ==
      classe(s) : --- C ---
   Cours A - 3
      local : #( 2 )#
      prof(s) : == 2 ==, == 4 ==
      classe(s) : --- A ---
   Conferences
      local : #( 1 )#
      prof(s) : -
      classe(s) :
=================================================
Toutes les possibilites ont ete explorees.
Temps total de recherche     : 524.45 s CPU
Nombre de solutions trouvees : 2
```

Fichier horInfo

```
%
%         horInfo   (probleme reel)
%
%         Horaire de l'Institut d'Informatique,
%                   cycle de 3 ans, 2d semestre
%
%               * modele de base (avec indisponibilites reelles)
%               * plages imposees
%               * cours simultanes
%               * pas de cours successifs
%               * locaux imposes


%====== grille ======

lu1  est_la_plage  'lundi de 8h30 a 10h30'.
lu2  est_la_plage  'lundi de 10h40 a 12h40'.
lu3  est_la_plage  'lundi de 14h00 a 16h00'.
lu4  est_la_plage  'lundi de 16h00 a 18h00'.
ma1  est_la_plage  'mardi de 8h30 a 10h30'.
ma2  est_la_plage  'mardi de 10h40 a 12h40'.
ma3  est_la_plage  'mardi de 14h00 a 16h00'.
ma4  est_la_plage  'mardi de 17h00 a 19h00'.
me1  est_la_plage  'mercredi de 8h30 a 10h30'.
me2  est_la_plage  'mercredi de 10h40 a 12h40'.
me3  est_la_plage  'mercredi de 14h00 a 16h00'.
me4  est_la_plage  'mercredi de 16h00 a 18h00'.
je1  est_la_plage  'jeudi de 8h30 a 10h30'.
je2  est_la_plage  'jeudi de 10h40 a 12h40'.
je3  est_la_plage  'jeudi de 14h00 a 16h00'.
je4  est_la_plage  'jeudi de 16h00 a 18h00'.
ve1  est_la_plage  'vendredi de 8h30 a 10h30'.
ve2  est_la_plage  'vendredi de 10h40 a 12h40'.
ve3  est_la_plage  'vendredi de 14h00 a 16h00'.
ve4  est_la_plage  'vendredi de 16h00 a 18h00'.



%====== locaux ======

i1 est_le_local 'Auditoire I1'.
i2 est_le_local 'Auditoire I2'.
i3 est_le_local 'Auditoire I3'.
s1 est_le_local 'Seminaire S1'.
s2 est_le_local 'Seminaire S2'.
s3 est_le_local 'Seminaire S3'.
s4 est_le_local 'Seminaire S4'.
s5 est_le_local 'Seminaire S5'.



%--- capacites ---

i1 a_une_capacite_de 60.
i2 a_une_capacite_de 146.
i3 a_une_capacite_de 60.
s1 a_une_capacite_de 40.
s2 a_une_capacite_de 40.
s3 a_une_capacite_de 40.
s4 a_une_capacite_de 40.
s5 a_une_capacite_de 40.
```

```
%====== professeurs ======

berl est_le_prof 'J. Berleur'.
boda est_le_prof 'F. Bodart'.
card est_le_prof 'J.P. Cardinael'.
caud est_le_prof 'R. Caudano'.
cher est_le_prof 'C. Cherton'.
coll est_le_prof 'Th. Collet-PetitJean'.
detr est_le_prof 'B. Detrembleur'.
drab est_le_prof 'J. Drabs'.
fich est_le_prof 'J. Fichefet'.
gigo est_le_prof 'R. Gigot'.
hain est_le_prof 'J.L. Hainaut'.
jamo est_le_prof 'N. Jamotte-Dachouffe'.
lech est_le_prof 'B. Le Charlier'.
lero est_le_prof 'H. Leroy'.
lesu est_le_prof 'R. Lesuisse'.
osty est_le_prof 'P. Ostyn'.
stas est_le_prof 'C. Stas-Mahiat'.
noir est_le_prof 'M. Noirhomme-Fraiture'.
nize est_le_prof 'J. Nizet'.
rama est_le_prof 'J. Ramaekers'.
vanb est_le_prof 'Ph. van Bastelaer'.
vanl est_le_prof 'A. van Lamsweerde'.



%--- indisponibilites

berl est_indisponible_pour lu1.
berl est_indisponible_pour lu2.
berl est_indisponible_pour lu3.
berl est_indisponible_pour lu4.
berl est_indisponible_pour ma1.
berl est_indisponible_pour ma2.
berl est_indisponible_pour ma3.
berl est_indisponible_pour ma4.
berl est_indisponible_pour je1.
berl est_indisponible_pour je2.
berl est_indisponible_pour je3.
berl est_indisponible_pour je4.
berl est_indisponible_pour ve1.
berl est_indisponible_pour ve2.
berl est_indisponible_pour ve3.
berl est_indisponible_pour ve4.

boda est_indisponible_pour lu1.
boda est_indisponible_pour lu2.
boda est_indisponible_pour lu3.
boda est_indisponible_pour lu4.
boda est_indisponible_pour ma1.
boda est_indisponible_pour ma2.
boda est_indisponible_pour ma3.
boda est_indisponible_pour ma4.
boda est_indisponible_pour me1.
boda est_indisponible_pour me2.
boda est_indisponible_pour me3.
boda est_indisponible_pour me4.
boda est_indisponible_pour ve1.
```

```
boda est_indisponible_pour ve2.

%- card    (un seul cours, plage imposee)

caud est_indisponible_pour ma4.

cher est_indisponible_pour me3.
cher est_indisponible_pour me4.

%- coll

%- detr

drab est_indisponible_pour lu2.
drab est_indisponible_pour lu3.
drab est_indisponible_pour lu4.
drab est_indisponible_pour ma2.
drab est_indisponible_pour ma3.
drab est_indisponible_pour ma4.
drab est_indisponible_pour me2.
drab est_indisponible_pour me3.
drab est_indisponible_pour me4.
drab est_indisponible_pour je1.
drab est_indisponible_pour je2.
drab est_indisponible_pour je3.
drab est_indisponible_pour je4.
drab est_indisponible_pour ve2.
drab est_indisponible_pour ve3.
drab est_indisponible_pour ve4.

fich est_indisponible_pour lu1.
fich est_indisponible_pour lu2.
fich est_indisponible_pour lu3.
fich est_indisponible_pour lu4.
fich est_indisponible_pour ma3.
fich est_indisponible_pour ma4.
fich est_indisponible_pour me3.
fich est_indisponible_pour me4.
fich est_indisponible_pour je3.
fich est_indisponible_pour je4.
fich est_indisponible_pour ve3.
fich est_indisponible_pour ve4.

gigo est_indisponible_pour je1.
gigo est_indisponible_pour je2.
gigo est_indisponible_pour je3.
gigo est_indisponible_pour je4.
gigo est_indisponible_pour ve1.
gigo est_indisponible_pour ve2.
gigo est_indisponible_pour ve3.

hain est_indisponible_pour lu1.
hain est_indisponible_pour lu2.
hain est_indisponible_pour lu3.
hain est_indisponible_pour lu4.
hain est_indisponible_pour ma1.
hain est_indisponible_pour ma2.
hain est_indisponible_pour ma3.
hain est_indisponible_pour ma4.
hain est_indisponible_pour ve2.

%- jamo
```

```
lech est_indisponible_pour lu1.
lech est_indisponible_pour ma3.
lech est_indisponible_pour me1.
lech est_indisponible_pour me2.
lech est_indisponible_pour me3.
lech est_indisponible_pour me4.
lech est_indisponible_pour ve1.
lech est_indisponible_pour ve2.
lech est_indisponible_pour ve3.
lech est_indisponible_pour ve4.

%- lero

lesu est_indisponible_pour lu1.
lesu est_indisponible_pour lu4.
lesu est_indisponible_pour ma1.
lesu est_indisponible_pour ma4.
lesu est_indisponible_pour ma2.
lesu est_indisponible_pour me1.
lesu est_indisponible_pour me2.
lesu est_indisponible_pour me3.
lesu est_indisponible_pour me4.
lesu est_indisponible_pour je1.
lesu est_indisponible_pour je4.
lesu est_indisponible_pour ve1.
lesu est_indisponible_pour ve2.
lesu est_indisponible_pour ve4.

%- osty

%- stas

nize est_indisponible_pour me1.
nize est_indisponible_pour me4.
nize est_indisponible_pour ve1.
nize est_indisponible_pour ve2.
nize est_indisponible_pour ve3.
nize est_indisponible_pour ve4.

noir est_indisponible_pour lu1.
noir est_indisponible_pour me3.
noir est_indisponible_pour me4.
noir est_indisponible_pour ve1.
noir est_indisponible_pour ve2.
noir est_indisponible_pour ve3.
noir est_indisponible_pour ve4.

rama est_indisponible_pour ma2.
rama est_indisponible_pour ma3.

vanb est_indisponible_pour lu3.
vanb est_indisponible_pour lu4.
vanb est_indisponible_pour ma1.
vanb est_indisponible_pour ma3.
vanb est_indisponible_pour ma4.
vanb est_indisponible_pour me1.
vanb est_indisponible_pour me3.
vanb est_indisponible_pour me4.
vanb est_indisponible_pour je1.
vanb est_indisponible_pour je3.
vanb est_indisponible_pour je4.
```

```
vanb est_indisponible_pour ve1.
vanb est_indisponible_pour ve2.
vanb est_indisponible_pour ve3.
vanb est_indisponible_pour ve4.

vanl est_indisponible_pour lu1.
vanl est_indisponible_pour lu2.
vanl est_indisponible_pour lu3.
vanl est_indisponible_pour lu4.
vanl est_indisponible_pour ma1.
vanl est_indisponible_pour ma4.
vanl est_indisponible_pour me1.
vanl est_indisponible_pour me4.
vanl est_indisponible_pour je1.
vanl est_indisponible_pour je2.
vanl est_indisponible_pour je3.
vanl est_indisponible_pour je4.
vanl est_indisponible_pour ve1.
vanl est_indisponible_pour ve2.
vanl est_indisponible_pour ve3.
vanl est_indisponible_pour ve4.



%====== classes ======

111 est_la_classe '1e licence - serie 1'.
112 est_la_classe '1e licence - serie 2'.
12s est_la_classe '2e licence (SI)'.
12m est_la_classe '2e licence (ML)'.
13s est_la_classe '3e licence (SI)'.
13m est_la_classe '3e licence (ML)'.



%--- effectifs ---
111 a_un_effectif_de 32.
112 a_un_effectif_de 32.
12m a_un_effectif_de 25.
12s a_un_effectif_de 25.
13m a_un_effectif_de  7.
13s a_un_effectif_de 28.



%====== cours ======

%=== 1ere licence ===

cbd est_le_cours 'Conception de bases de donnees'.
cob est_le_cours 'Programmation d''applications de gestion en langage Cobol'.
csi est_le_cours 'Conception des systemes d''information'.
lg1 est_le_cours 'Anglais-americain et neerlandais'.
mpr est_le_cours 'Methodologie de la programmation'.
pst est_le_cours 'Processus stochastiques'.
tin est_le_cours 'Physique mathematique (theorie de l''information)'.
tor est_le_cours 'Theorie des organisations'.

tp_csi_1 est_le_cours 'T.P. Conception des systemes d''information - serie 1'.
tp_csi_2 est_le_cours 'T.P. Conception des systemes d''information - serie 2'.
tp_cob_1 est_le_cours 'T.P. Cobol - serie 1'.
tp_cob_2 est_le_cours 'T.P. Cobol - serie 2'.
```

```
tp_sde   est_le_cours 'T.P. Concepts des systemes d''exploitation'.

recup_1  est_le_cours 'Plage de recuperation - 1ere licence'.


%--- cours/prof

cbd est_donne_par [hain].
cob est_donne_par [lesu].
csi est_donne_par [boda].
lg1 est_donne_par [osty].
mpr est_donne_par [lero].
pst est_donne_par [noir].
tin est_donne_par [caud].
tor est_donne_par [drab].
tp_csi_1 est_donne_par [coll].
tp_csi_2 est_donne_par [coll].
tp_cob_1 est_donne_par [jamo].
tp_cob_2 est_donne_par [jamo].
tp_sde   est_donne_par [rama,detr,stas].
recup_1  est_donne_par [].


%--- cours/classe

cbd est_suivi_par [111,112].
cob est_suivi_par [111,112].
csi est_suivi_par [111,112].
lg1 est_suivi_par [111,112].
mpr est_suivi_par [111,112].
pst est_suivi_par [111,112].
tin est_suivi_par [111,112].
tor est_suivi_par [111,112].
tp_csi_1 est_suivi_par [111].
tp_csi_2 est_suivi_par [112].
tp_cob_1 est_suivi_par [111].
tp_cob_2 est_suivi_par [112].
tp_sde   est_suivi_par [111,112].
recup_1  est_suivi_par [111,112].


%=== 2eme licence ===

amu est_le_cours 'Analyse multicritere'.
cec est_le_cours 'Calculabilite et complexite'.
com est_le_cours 'Compilation (m.a.)'.
csi2 est_le_cours 'Conception des systemes d''information (m.a.)'.
ihm est_le_cours 'Interfaces homme/machine'.
lg2 est_le_cours 'Neerlandais'.
mdp est_le_cours 'Mesures de performances'.
mep est_le_cours 'Modeles et evaluation de performances'.
mpr2 est_le_cours 'Methodologie de la programmation (m.a.)'.
psi est_le_cours 'Aspects psychologiques des systemes d''information'.
sde3 est_le_cours 'Systemes d''exploitation, etude de cas'.
sib est_le_cours 'Systemes d''information du bureau et d''aide a la decision'.
sta est_le_cours 'Statistique appliquee'.
tin2 est_le_cours 'Teleinformatique (m.a.)'.
tor2 est_le_cours 'Theorie des organisations (m.a.)'.
tpr est_le_cours 'Theorie des programmes'.
recup_2  est_le_cours 'Plage de recuperation - 2eme licence'.
```

```
%--- cours/prof

amu  est_donne_par [fich].
cec  est_donne_par [lero].
com  est_donne_par [lero].
csi2 est_donne_par [boda].
ihm  est_donne_par [boda].
lg2  est_donne_par [osty].
mdp  est_donne_par [rama].
mep  est_donne_par [noir].
mpr2 est_donne_par [cher].
psi  est_donne_par [nize].
sde3 est_donne_par [rama].
sib  est_donne_par [lesu].
sta  est_donne_par [noir].
tin2 est_donne_par [vanb].
tor2 est_donne_par [lesu].
tpr  est_donne_par [lech].
recup_2  est_donne_par [].


%--- cours/classe

amu  est_suivi_par [l2s].
cec  est_suivi_par [l2s,l2m].
com  est_suivi_par [l2m].
csi2 est_suivi_par [l2s,l2m].
ihm  est_suivi_par [l2s,l2m].
lg2  est_suivi_par [l2s,l2m].
mdp  est_suivi_par [l2m].
mep  est_suivi_par [l2m].
mpr2 est_suivi_par [l2m].
psi  est_suivi_par [l2s].
sde3 est_suivi_par [l2m].
sib  est_suivi_par [l2s,l2m].
sta  est_suivi_par [l2s,l2m].
tin2 est_suivi_par [l2m].
tor2 est_suivi_par [l2s].
tpr  est_suivi_par [l2m].
recup_2  est_suivi_par [l2s,l2m].


%=== 3eme licence ===

aad  est_le_cours 'Aide a la decision : etude de cas'.
csi3 est_le_cours 'Conception de systemes informatiques : etude de cas'.
csr  est_le_cours 'Conception de systemes repartis'.
eap  est_le_cours 'Etude d''une application a l''entrepr. ou a l''administr.'.
gri  est_le_cours 'Gestion des ressources informatiques'.
gsi  est_le_cours 'Gestion strategique des systemes d''information'.
iso  est_le_cours 'Seminaire d''"Informatique et societe"'.
lg3  est_le_cours 'Anglais et neerlandais'.
md12 est_le_cours 'Methodologie de developpement de logiciels (m.a.)'.
rog2 est_le_cours 'Recherche operationnelle et de gestion (m.a.)'.
sad  est_le_cours 'Systemes d''aide a la decision (m.a.)'.
sfs  est_le_cours 'Securite et fiabilite des systemes informatiques'.
sms  est_le_cours 'Simulation de systemes'.
tbd2 est_le_cours 'Technologie des bases de donnees (m.a.)'.
recup_3  est_le_cours 'Plage de recuperation - 3ere licence'.


%--- cours/prof
```

```
aad  est_donne_par [fich].
csi3 est_donne_par [boda].
csr  est_donne_par [vanl].
eap  est_donne_par [gigo].
gri  est_donne_par [card].
gsi  est_donne_par [boda].
iso  est_donne_par [berl].
lg3  est_donne_par [osty].
md12 est_donne_par [vanl].
rog2 est_donne_par [fich].
sad  est_donne_par [boda].
sfs  est_donne_par [rama].
sms  est_donne_par [noir].
tbd2 est_donne_par [hain].
recup_2  est_donne_par [].


%--- cours/classe

aad  est_suivi_par [13s].
csi3 est_suivi_par [13m].
csr  est_suivi_par [13m].
eap  est_suivi_par [13s].
gri  est_suivi_par [13s,13m].
gsi  est_suivi_par [13s].
iso  est_suivi_par [13s,13m].
lg3  est_suivi_par [13s,13m].
md12 est_suivi_par [13s,13m].
rog2 est_suivi_par [13s].
sad  est_suivi_par [13s].
sfs  est_suivi_par [13m].
sms  est_suivi_par [13m].
tbd2 est_suivi_par [13s,13m].
recup_2  est_suivi_par [13m,13s].


%====== Contraintes ======

cbd doit_etre_donne_en_plage ve3.     % Cours avec des economistes
lg1 doit_etre_donne_en_plage ma2.     % Reservation imposee
lg2 doit_etre_donne_en_plage ma1.     % Reservation imposee
lg3 doit_etre_donne_en_plage lu3.
gri doit_etre_donne_en_plage lu4.
csr doit_etre_donne_en_plage ma2.
md12 doit_etre_donne_en_plage ma3.


tp_csi_1 est_dans_le_groupe tp_series_a.
tp_cob_2 est_dans_le_groupe tp_series_a.

tp_csi_2 est_dans_le_groupe tp_series_b.
tp_cob_1 est_dans_le_groupe tp_series_b.


iso est_localise_dans s5.

%================================================================================
```

Fichier horI_LT1

```
**********************************************************
  Resolution du probleme decrit dans le fichier
                    horInfo
          (avec assignation de locaux)
----------------------------------------------
  Tri preliminaire des cours : EFFECTUE
  Temps de preparation : 45.9001 s  CPU.
  Temps minimal de recherche : 1 s  CPU.

**********************************************************
   Solution trouvee apres 103.4 secondes CPU
----------------------------------------------
Cours prevus pour  lundi de 8h30 a 10h30
   Securite et fiabilite des systemes informatiques
      local : Seminaire S2
      prof(s) : J. Ramaekers
      classe(s) : 3e licence (ML)
   Aspects psychologiques des systemes d'information
      local : Seminaire S1
      prof(s) : J. Nizet
      classe(s) : 2e licence (SI)
   Etude d'une application a l'entrepr. ou a l'administr.
      local : Auditoire I3
      prof(s) : R. Gigot
      classe(s) : 3e licence (SI)
   Teleinformatique (m.a.)
      local : Auditoire I1
      prof(s) : Ph. van Bastelaer
      classe(s) : 2e licence (ML)
   Theorie des organisations
      local : Auditoire I2
      prof(s) : J. Drabs
      classe(s) : 1e licence - serie 1, 1e licence - serie 2
Cours prevus pour  lundi de 10h40 a 12h40
   Simulation de systemes
      local : Auditoire I3
      prof(s) : M. Noirhomme-Fraiture
      classe(s) : 3e licence (ML)
   Theorie des programmes
      local : Auditoire I1
      prof(s) : B. Le Charlier
      classe(s) : 2e licence (ML)
   Programmation d'applications de gestion en langage Cobol
      local : Auditoire I2
      prof(s) : R. Lesuisse
      classe(s) : 1e licence - serie 1, 1e licence - serie 2
Cours prevus pour  lundi de 14h00 a 16h00
   T.P. Conception des systemes d'information - serie 2
      local : Seminaire S1
      prof(s) : Th. Collet-PetitJean
      classe(s) : 1e licence - serie 2
   T.P. Cobol - serie 1
      local : Auditoire I3
      prof(s) : N. Jamotte-Dachouffe
      classe(s) : 1e licence - serie 1
   Systemes d'information du bureau et d'aide a la decision
      local : Auditoire I2
      prof(s) : R. Lesuisse
      classe(s) : 2e licence (SI), 2e licence (ML)
   Anglais et neerlandais
      local : Auditoire I1
      prof(s) : P. Ostyn
```

classe(s) : 3e licence (SI), 3e licence (ML)
Cours prevus pour lundi de 16h00 a 18h00
T.P. Conception des systemes d'information - serie 1
local : Seminaire S1
prof(s) : Th. Collet-PetitJean
classe(s) : 1e licence - serie 1
T.P. Cobol - serie 2
local : Auditoire I3
prof(s) : N. Jamotte-Dachouffe
classe(s) : 1e licence - serie 2
Modeles et evaluation de performances
local : Auditoire I2
prof(s) : M. Noirhomme-Fraiture
classe(s) : 2e licence (ML)
Gestion des ressources informatiques
local : Auditoire I1
prof(s) : J.P. Cardinael
classe(s) : 3e licence (SI), 3e licence (ML)
Cours prevus pour mardi de 8h30 a 10h30
Aide a la decision : etude de cas
local : Auditoire I2
prof(s) : J. Fichefet
classe(s) : 3e licence (SI)
Neerlandais
local : Auditoire I1
prof(s) : P. Ostyn
classe(s) : 2e licence (SI), 2e licence (ML)
Cours prevus pour mardi de 10h40 a 12h40
Methodologie de la programmation (m.a.)
local : Seminaire S1
prof(s) : C. Cherton
classe(s) : 2e licence (ML)
Analyse multicritere
local : Auditoire I3
prof(s) : J. Fichefet
classe(s) : 2e licence (SI)
Anglais-americain et neerlandais
local : Auditoire I2
prof(s) : P. Ostyn
classe(s) : 1e licence - serie 1, 1e licence - serie 2
Conception de systemes repartis
local : Auditoire I1
prof(s) : A. van Lamsweerde
classe(s) : 3e licence (ML)
Cours prevus pour mardi de 14h00 a 16h00
Compilation (m.a.)
local : Auditoire I3
prof(s) : H. Leroy
classe(s) : 2e licence (ML)
Theorie des organisations (m.a.)
local : Auditoire I2
prof(s) : R. Lesuisse
classe(s) : 2e licence (SI)
Methodologie de developpement de logiciels (m.a.)
local : Auditoire I1
prof(s) : A. van Lamsweerde
classe(s) : 3e licence (SI), 3e licence (ML)
Cours prevus pour mardi de 17h00 a 19h00
Mesures de performances
local : Auditoire I1
prof(s) : J. Ramaekers
classe(s) : 2e licence (ML)

Processus stochastiques
local : Auditoire I2
prof(s) : M. Noirhomme-Fraiture
classe(s) : 1e licence - serie 1, 1e licence - serie 2
Cours prevus pour mercredi de 8h30 a 10h30
T.P. Concepts des systemes d'exploitation
local : Auditoire I2
prof(s) : J. Ramaekers, B. Detrembleur, C. Stas-Mahiat
classe(s) : 1e licence - serie 1, 1e licence - serie 2
Statistique appliquee
local : Auditoire I1
prof(s) : M. Noirhomme-Fraiture
classe(s) : 2e licence (SI), 2e licence (ML)
Seminaire d'"Informatique et societe"
local : Seminaire S5
prof(s) : J. Berleur
classe(s) : 3e licence (SI), 3e licence (ML)
Cours prevus pour mercredi de 10h40 a 12h40
Systemes d'exploitation, etude de cas
local : Auditoire I2
prof(s) : J. Ramaekers
classe(s) : 2e licence (ML)
Recherche operationnelle et de gestion (m.a.)
local : Auditoire I1
prof(s) : J. Fichefet
classe(s) : 3e licence (SI)
Cours prevus pour mercredi de 14h00 a 16h00
Calculabilite et complexite
local : Auditoire I3
prof(s) : H. Leroy
classe(s) : 2e licence (SI), 2e licence (ML)
Physique mathematique (theorie de l'information)
local : Auditoire I2
prof(s) : R. Caudano
classe(s) : 1e licence - serie 1, 1e licence - serie 2
Technologie des bases de donnees (m.a.)
local : Auditoire I1
prof(s) : J.L. Hainaut
classe(s) : 3e licence (SI), 3e licence (ML)
Cours prevus pour mercredi de 16h00 a 18h00
Plage de recuperation - 2eme licence
local : Auditoire I1
prof(s) : -
classe(s) : 2e licence (SI), 2e licence (ML)
Plage de recuperation - 2eme licence
local : Auditoire I1
prof(s) : -
classe(s) : 3e licence (ML), 3e licence (SI)
Plage de recuperation - 2eme licence
local : Auditoire I1
prof(s) : -
classe(s) : 2e licence (SI), 2e licence (ML)
Plage de recuperation - 2eme licence
local : Auditoire I1
prof(s) : -
classe(s) : 3e licence (ML), 3e licence (SI)
Methodologie de la programmation
local : Auditoire I2
prof(s) : H. Leroy
classe(s) : 1e licence - serie 1, 1e licence - serie 2
Cours prevus pour jeudi de 8h30 a 10h30
Conception des systemes d'information

```
      local : Auditoire I2
      prof(s) : F. Bodart
      classe(s) : 1e licence - serie 1, 1e licence - serie 2
Cours prevus pour  jeudi de 10h40 a 12h40
   Plage de recuperation - 1ere licence
      local : Auditoire I2
      prof(s) : -
      classe(s) : 1e licence - serie 1, 1e licence - serie 2
   Conception des systemes d'information (m.a.)
      local : Auditoire I1
      prof(s) : F. Bodart
      classe(s) : 2e licence (SI), 2e licence (ML)
Cours prevus pour  jeudi de 14h00 a 16h00
   Conception de systemes informatiques : etude de cas
      local : Auditoire I1
      prof(s) : F. Bodart
      classe(s) : 3e licence (ML)
Cours prevus pour  jeudi de 16h00 a 18h00
   Gestion strategique des systemes d'information
      local : Auditoire I1
      prof(s) : F. Bodart
      classe(s) : 3e licence (SI)
Cours prevus pour  vendredi de 8h30 a 10h30
Cours prevus pour  vendredi de 10h40 a 12h40
Cours prevus pour  vendredi de 14h00 a 16h00
   Interfaces homme/machine
      local : Auditoire I1
      prof(s) : F. Bodart
      classe(s) : 2e licence (SI), 2e licence (ML)
   Conception de bases de donnees
      local : Auditoire I2
      prof(s) : J.L. Hainaut
      classe(s) : 1e licence - serie 1, 1e licence - serie 2
Cours prevus pour  vendredi de 16h00 a 18h00
   Systemes d'aide a la decision (m.a.)
      local : Auditoire I1
      prof(s) : F. Bodart
      classe(s) : 3e licence (SI)
================================================
Le temps minimal de recherche est atteint.
Nombre de solutions trouvees : 1
```